Model Order Reduction of Rarefied Gases using Neural Networks

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
im Fach Physikalische Ingenieurswissenschaften

Technische Universität Berlin
Fakultät Verkehrs- und Maschinensysteme V
Institut für Numerische Fluiddynamik

eingereicht von: *Zachary Schellin*
geboren am: *11.02.1991, Berlin*

Gutachter: *Prof. Dr. Julius Reiss*
Dr. Mathias Lemke

eingereicht am: *01. Juli 2021*

Abstract

Autoencoders for model order reduction (MOR) in rarefied gasdynamics are tested on the BGK model in Sod's shock tube for two levels of rarefaction with proper orthogonal decomposition (POD) as a benchmark. Therefor, a convolutional neural network (CNN) as well as a fully connected neural network (FCNN) are designed in terms of finding appropriate hyperparameters for the given task. The FCNN supersedes POD only in the number of parameters used to achieve a reconstruction loss above 1×10^{-2} for the two rarefaction levels. The CNN fails to reproduce usable results, giving rise to further insight into chosen hyperparameters. Interpretabillity of the intrinsic variables extracted from the FCNN is given to a certain extend. Similarities with macroscopic quantities are present for both flows, though only entangled.

Zusammenfassung

german abstrac here

Contents

1 Introduction

The Bhatnagar-Gross-Krook equation (BGK) is a kinetic collision model of ionized and neutral gases valid for rarefied as well as other pressure regimes [1]. Generating data of such a flow field is essential for various industry and scientific applications[**REF**]. With the intention to reduce time and cost during the data generating process, experiments were substituted with computational fluid dynamics (CFD) computations. Consequently reduced-order models (ROMs) coupled to aforementioned computations were introduced to further the reduction of time and cost. The thriving field of artificial intelligence successfully operates on natural language processing and object recognition and has now surfaced in fluid mechanics for model order reduction as seen in [2] and [3]. This thesis wants to estimate the capability of artifical intelligence for model order reduction in fluid mechanic by specifically revising the performance of autoencoders in a fully connected and convolutional version using proper orthogonal decomposition (POD) as a benchmark. Thereto the BGK model in Sod's shock tube is employed.

1.1 Thesis outline

The thesis is structured in five sections. First the BGk model in Sod's sock tube is introduced in chapter 2. Furthermore, dimensionality reduction algorithms used in this thesis namely proper orthogonal decomposition (POD) and autoencoders are introduced in chapter 3. Additionally model order reduction and the integration of POD and autoencoders with emphasis on the so called offline phase is described in chapter 4. The comparison of both methods by evaluating their reconstruction loss and the interpretability of the so called reduced variables can be found in chapter 5. In addition the autoencoders abillity to generalize is also tested in this section. A description covering the selection of design features for the convolutional and fully connected autoencoder is available in appendix A. The code for this thesis is written in `Python` version 3.8. Both autoencoders are implemented using the open source machine learning library `pyTorch` version 1.8.1 developed by Facebook's AI Reasearch lab (FAIR) which is available from `www.pytorch.org`. Furthermore `NumPy` for any additional computations aside the machine learning aspect and `pandas` for data manipulation available through the open source scientific computing library `SciPy` version 1.6.3 available through `www.scipy.org` is used.

1.2 State of the art

State of the art model reduction of dynamical systems can be done via proper orthogonal decomposition (POD) which is an algorithm feeding on the idea of singular value decomposition (SVD)[4][5]. POD captures a low-rank representation on a linear manifold. So called POD modes,

derived from SVD, describe the principle components of a problem which can be coupled within a Galerkin framework to produce an approximation of a lower dimension r. Bernard et al. use POD-Galerkin with an additional population of their snapshot database via optimal transport for the proposed BGK equation, bisecting computational run time (cost) in conjunction with an approximation error of ~ 1 % in [6]. Artificial intelligence in the form of autoencoders replacing the POD within a Galerkin framework is evaluated against the POD performance by Kookjin et al. for advection-dominated problems[2] resulting in sub 0.1% errors. An additional time inter- and extrapolation is evaluated. Using machine learning/ deep learning for reduced order modeling in CFD is a novel approach although "the idea of autoencoders has been part of the historical landscape of neural networks for decades"[7, p.493]. Autoencoders, or more precisely learning internal representations by the delta rule (backpropagation) and the use of hidden units in a feed forward neural network architecture, premiered by Rumelhart et al. (1986) [8]. Through so called hierarchical training Ballard et al.(1987) introduce a strategy to train auto autoassociative networks (nowadays referred to as autoencoders), in a reasonable time promoting further development despite computational limitations [9]. The so called bottleneck of autoencoders yields a non-smooth and entangled representation thus beeing uninterpretable by practitioners[10] leading to developements in this field. Rifai et al. introduce the contractive autoencoder (CAE) for classification tasks (2011), with the aim to extract robust features which are insensitive to input variations orthogonal to the low-dimensional non-linear manifold by adding a penalty on the frobenius norm of the instrinsic variables with respect to the input, surpassing other classification algorithms [10]. Subsequent development emerges with the manifold tangent classifier (MTC) [11]. A local chart for each datapoint is obtained hence characterizing the manifold which in turn improves classification performance. On that basis a generative process for the CAE is developed. Through movements along the manifold with directions defined by the Jacobian of the bottleneck layer with respect to the input $\vec{x}_m = JJ^T$, sampling is realized [12].... Proper orthogonal decomposition (POD) and it's numerous variants like shifted-POD[**?**], POD-Galerkin[**?**], POD+I [**?**] to name only a few of them, try to solve this problem by.....

2 The BGK Model

This chapter covers the BGK model, a kinetic gas model in Sod's shock tube. Sod's shock tube serves here as a test case for which solutions of the BGK model were made available for this thesis. The BGK model is valid for a broad range of rarefaction levels. Ranging from continuum flows, where the Navier-Stokes equations can be utilized to highly rarefied regimes. Rarefaction levels are labeled with the so called Knudsen number Kn first introduced by danish physicist Martin Knudsen [6]. The Knudsen number is given by

$$Kn = \frac{\lambda}{l}, \tag{2.1}$$

where λ is the mean free path of a particle and l, some domain specific length as the diameter of a tube for example. Figure 2.1 shows a possible partitioning of Kn into specific regimes, which is quiet comfortable. Though no clear cut can be applied to different values of Kn. The boundaries are particularly blurry [13].

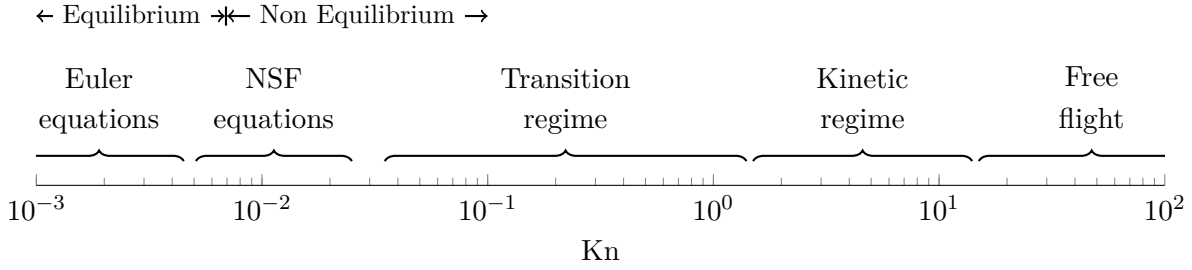


Figure 2.1: Partitioning of Kn, the Knudsen number, into levels of rarefaction. The Euler equations can be used up to approximately $Kn < 0.01$ and describe a „continuum flow". A "slip flow" can be defined in the interval $0.01 < Kn < 0.1$, termed as slightly rarefied in [13]. Here the Navier-stokes-Fourrier equations yield accurate results. From $Kn > 0.1$ onwards (transition regime, kinetic regime and free flight) the rarefaction increases steadily and only kinetic gas models deliver reasonable computations. The BGK model can be used for all rarefaction levels.

2.1 Space and Velocity Discretization, Moments and Conservation

The BGK model was introduced by, and named after, physicists Prabhu L. Bhatnagar, Eugene P. Gross and Max Krook in 1954 [1]. It is an approximation of the standard Boltzmann transport equation. More precisely the right hand side of the Boltzmann equation is approximated by the BGK operator

$$\partial_t f + v\partial_x f = \frac{1}{\tau}(M_f - f), \tag{2.2}$$

which can be found in [14].It features the relaxation time $\tau(x,t)$, the Maxwellian distribution M_f and $f(t,v,x)$ the probability of a gas particle having a microscopic velocity v in phase space (t,v,x).

The right hand side is a source term describing the distance between the current probability density function f and it's equilibrium solution M_f. Evidently when $f = M_f$ the right hand side becomes zero. More precisely equilibirum is reached. A time scale for which f transitions into equilibrium is given by τ which can be defined with

$$\tau^{-1} = \frac{\rho(x,t)T^{1-\nu}(x,t)}{Kn}, \tag{2.3}$$

taken from [6]. Through τ the viscosity exponent ν, density ρ, temperature T and Knudsen number Kn additionally establish a scaling factor for the right hand side of the BGK model. The Maxwellian distribution M_f is defined by $\rho(x,t)$, $T(x,t)$ and $u(x,t)$ the macroscopic velocity

$$M_f = \frac{\rho(x,t)}{(2\pi RT(x,t))^{\frac{3}{2}}} \exp(-\frac{(v-u(x,t))^2}{2RT(x,t)}). \tag{2.4}$$

The left hand side of the BGK model is the Boltzmann transport equation for $f(t,v,x)$ with microscopic transport velocity v. In one dimension, the BGK model needs to be evaluated for the three independent variables x, v and t as seen above. Furthermore in three dimensions one needs to include the evaluation at (x,y,z) in space and (v_x,v_y,v_z) in velocity space. Note that in this thesis the BGK model is discussed in one dimension only.
Now what makes the BGK model especially attractive for model order reduction? The fruitfullness of performing a model order reduction on the BGK model becomes clearer when looking at it's space and velocity discretization

$$\partial_t f_{j,k} = -(v_k)_1 D_x f|_{j,k}(t) + \frac{1}{\tau}(M_{f\,j,k}(t) - f_{j,k}(t)), \tag{2.5}$$

found in [14]. Here a uniform grid is considered with $x_j = j\Delta x$, $j \in \mathbb{Z}$, $v_k = k\Delta v$, $k \in \mathbb{Z}$ and $t^i = i\Delta t$, $i \in \mathbb{N}$ on which $f_{j,k} = f(t, v_k, x_j)$ and $M_{f\,j,k} = M_f(x_j, v_k, t)$ are evaluated at point (x_j, v_k) in a time instance t. For brevity $D_x f|_{j,k}$ is the discrete space derivative at (x_j, v_k). Now the partial differential equation (PDE) in eq. (2.2) is broken down into a system of ordinary differential equations (ODE's) in time, for which every ODE is a linear advection equation with constant scalar speed v_k and a source term.

To continue let's consider K to be the number of gridpoints in velocity and J to be the number of grid points in space. Then a total of KJ first order differential equations need to be evaluated in 1D. Obviously in three dimensions the system of ODE's inflates up to K^3J^3 first order differential equations. This in turn drives the evaluation of the BGK model at the edge of intractability for dense meshes in 3D and all together motivate for a reduced order model.
A closer look on the discretization in velocity space yields the necessity to compute the moments of f and provides a system of conservative equations.

Moments or expected values of f are the density ρ, the momentum ρu and the energy E in velocity space which can be obtained with

$$\rho(x,t) = \int f \,\mathrm{d}v \quad (2.6) \quad , \quad \rho(x,t)u(x,t) = \int vf \,\mathrm{d}v \quad (2.7) \quad \text{and} \quad E(x,t) = \int \frac{1}{2}v^2 f \,\mathrm{d}v, \quad (2.8)$$

as explained in [14]. Additionally the temperature T and the pressure p can be obtained with

$$T = \frac{2E}{3\rho} - \frac{||u||^2}{3} \qquad (2.9) \qquad \text{and} \quad p = \rho T \qquad (2.10)$$

With multiplying $\Phi(v) = [1, v, \frac{1}{2}v^2]$, called the collision invariants, and integrating in velocity space, one obtains the moments of f, which are needed to compute the maxwellian in eq. (2.5).

Again the system in eq. (2.2) is in equilibrium when $f = M_f$. Now by multiplying the equilibrium solution (left hand side of eq. (2.2) substituting $f = M_f$) by $\Phi(v)$ and integrating in velocity space, one finds the Euler system of classical gas dynamics using the equation of state of the gas as in [14]. These are

$$\partial_t \rho + \partial_x(\rho u) = 0, \qquad (2.11)$$
$$\partial_t(\rho u) + \partial_x(\rho u^2 + p) = 0, \qquad (2.12)$$
$$\partial_t E + \partial_x(u(E + p)) = 0, \qquad (2.13)$$

and provide conservation laws for the BGK model. Note that the evaluation of the maxwellianM_f is not straight forward and requires three additional non linear equations for every K-th grid point in velocity space. This is due to the fact, that for M_f the moments in eq. (2.6), eq. (2.7) and eq. (2.8) are needed. The quadrature rule used to compute the moments requires to be exact because even small errors magnify when $\tau \to 0$ and in turn one fails to obtain the Euler equations. Therefore M_f must satisfy

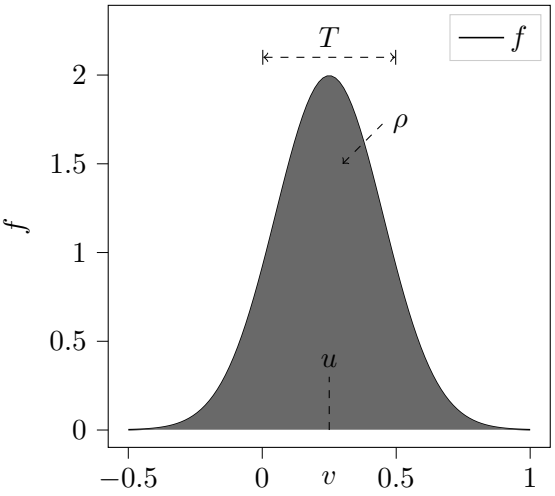


Figure 2.2: Illustration of the linkage between the macroscopic quantities of the gas flow and the distribution function f.

$$< \Phi(v) f > - < \Phi(v) M_f >= 0, \qquad (2.14)$$

which is accomplished by the computation of a discrete Maxwellian $\mathcal{M}_f$ by solving

$$\sum_k w_k \Phi(v_k)[f(t, v_k, x) - \exp(\alpha(x, t)\Phi(v_k))] = 0. \qquad (2.15)$$

Here w_k are weights and $\alpha(x, t)$ is a vector of three elements from which a unique solution for can be determined for $\mathcal{M}_f$. Further insights on $\mathcal{M}_f$ and time discretization will be omitted.

Displayed in fig. 2.2 is a demonstrative example of how the distribution function $f(v)$ gives the values for the macroscopic quantities. The distribution is centered around the macroscopic velocity u, the mean velocity of the distribution f is the temperature T, integrating $f(v)$ over velocity space one obtains the density ρ.

The BGK model inherits global conservation of mass, momentum and energy from the Boltzmann equation as seen in eq. (2.11), eq. (2.12) and eq. (2.13) found in [14].

2.2 Sod's shock tube as a test case for the BGK model

Using a shock tube as a test case for numerical schemes solving nonlinear hyperbolic conservation laws in gas dynamics was studied by Gary A. Sod in 1978 [15]. He evaluated different schemes in their performance of capturing the rarefaction wave, the contact discontinuity and the shock wave, which develop in the shock tube. Since then it serves as a commonly used benchmark problem in numerical gas dynamics.
Nonlinear conservation laws in a simple shock tube can be solved analytically and thereafter be compared to the numerical approximation. The analytical solution is obtained using the method of characteristics and the Rankine Hugoniot jump conditions to connect the states before and after the shock. Details about both methods can be found in [16].

The problem setup for a shock tube at $t = 0$ is shown in fig. 2.3 and fig. 2.4a, which is split into two regions (region 1 and region 5) via a diaphragm. Here the initial conditions for two fluids at rest are $\rho_0 = 1$ and $\rho_5 = 0.125$ for the density, $p_0 = 1$ and $p_5 = 0.1$ for the pressure and $u_0 = u_5 = 0$ for the macroscopic velocity [15].

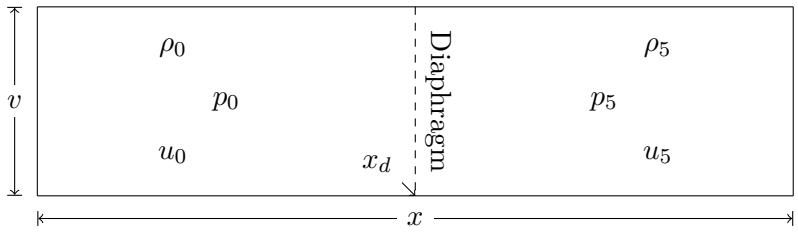
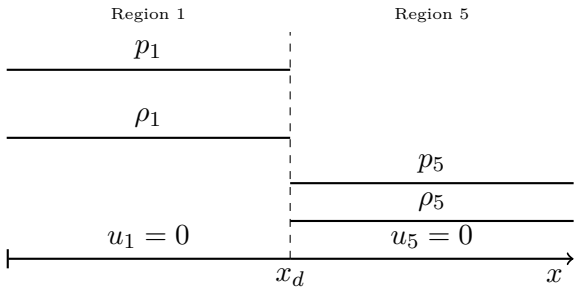
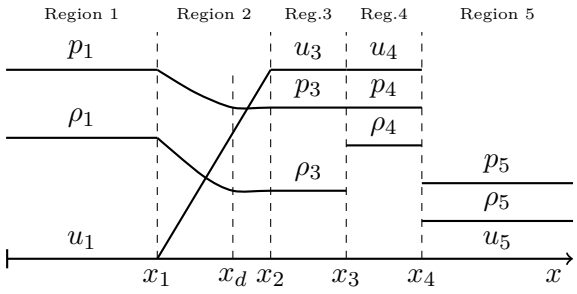


Figure 2.3: Problem setup for Sod's shock tube in 1D translated for the BGK model in velocity v and space x. A diaphragm is positioned at x_d, seperating the whole domain in two regions (region 1 and region 5). Initial conditions for density ρ, pressure p and macroscopic velocity u are indicated.

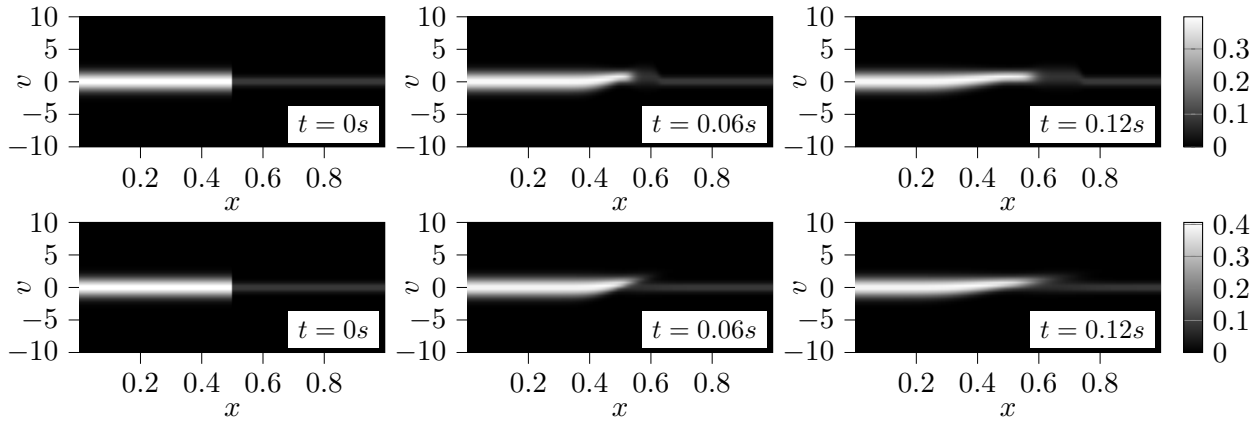
At $t > 0$ the diaphragm is broken, which leads to the formation of five regions which are depicted in fig. 2.4b. Between x_1 and x_2 we find the head and tail of the rarefaction wave traveling left. The solution for ρ, p and u is continuous in this area. The rarefaction wave is clearly discernible for a dilution of $Kn = 0.01$ and $Kn = 0.00001$ as seen fig. 2.4d. The contact discontinuity at x_3 is the point where a particle traveled from it's initial location at x_d in a time Δt. The original paper by Sod mentions here, that across the contact discontinuity x_3 the macroscopic velocity u and the pressure p are continuous in contrast to the density ρ and the energy E, as depicted in fig. 2.4b. This cannot be assumed for rarefied gases with $Kn = 0.01$ as seen in fig. 2.4d. A pronounced contact discontinuity in the density ρ and the energy E cannot be found. Labeled as x_4 is the position of the shock wave, at which in general none of the microscopic quantities will be continuous for gases with $Kn = 0.00001$. Again this does not hold for rarefied regimes as seen in fig. 2.4d.
Note, that fig. 2.4a and fig. 2.4b is taken from [15] in order to elaborate the general evolution in time of a gas of $Kn < 0.01$ in Sod's shock tube. Figure 2.4c shows solutions $f(t_i, v, x)$ of the BGK model with $t_0 = 0s$,$t_1 = 0.06$ and $t_3 = 0.12s$ for two rarefaction levels $Kn = 0.00001$ and $Kn = 0.01$. There the difference when increasing the dilution of a gas in Sod's shock tube is visible: An increased dilution leads to a smooth transition from region 1 to region 5 with the abundance of a pronounced shock front.



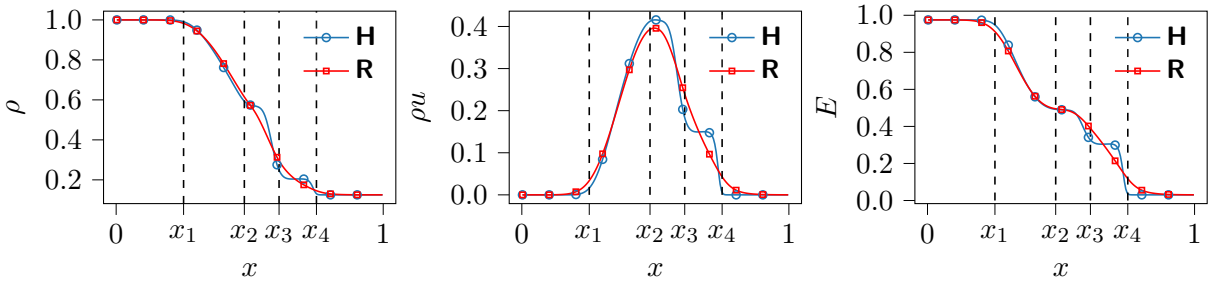
(a) Sod's shock tube at $t = 0$. The whole domain is split into two regions with corresponding initial conditions for pressure p, density ρ and macroscopic velocity u. Position of the diaphragm is labeled as x_d.



(b) Sod's shock tube at $t > 0$. Shown are pressure p, denisty ρ and macroscoppic velocity u. Five regions can be identified marked out with x_1 and x_2 as head and tail of the rarefaction wave, x_3 as the contact discontinuity and x_4 as the position of the shock wave. The position of the initial diaphragm is labeled x_d. A particle that traveled from x_d during Δt will be located at x_3.



(c) Two solutions of the BGK model $f(t_i, v, x)$ in Sod's shock tube with $Kn = 0.00001$ (top row) and $Kn = 0.01$ (bottom row) for a fixed time t_i. Solutions are presented for $t_0 = 0s$, $t_1 = 0.06s$ and $t_2 = 0.12s$.



(d) Macroscopic quantities $\rho(x, t_i)$, $\rho u(x, t_i)$ and $E(x, t_i)$ in the SOD schock tube at $t_i = 0.12s$. Displayed are the quantities for $Kn = 0.00001$ and for $Kn = 0.01$, where the former is abbreviated with **H** and the latter with **R**. The locations of head and tail of rarefaction wave x_1 and x_2, contact discontinuity x_3 and shockwave x_4 are labeled.

Figure 2.4: BGK model in Sod's shock tube: Initial conditions and their evolution after Δt are shown in (a) and (b). Two solutions of differing rarefaction levels are presented in (c). Macroscopic quantities of (c) at $t = 0.12s$ are shown in (d).

3 Dimensionality reduction algorithms

In this chapter dimensionality reduction algorithms which will be applied to solutions of the BGK model in Sod's shock tube are introduced: Proper orthogonal Decomposition (POD) and Autoencoders (AEs).
We will start off with a short introducing into POD, continue with autoencoders and finish with a detailed description of deep learning. Note that the main focus of this thesis lies in the application of autoencoders for which POD serves as a comparative method.
Dimensionality reduction algorithms lie at the heart of any reduced order model (ROM) which is described in the following chapter. As input serve datasets such as solutions of a full order model (FOM) or experimental data. These datasets may contain the dynamics of a spatio-temporal problem. The output is an approximation to the input. It is reconstructed from a low-dimensional representation that captured the underlying dynamics of the input problem.

3.1 Proper orthogonal decomposition (POD)

The solution of PDEs, precisely $f(x, v, t)$, can be approximated either through a discretization into a system of ODEs as described in chapter 2, or alternatively through a separation of variables ansatz

$$f(t, v, x) = \sum_{i=1}^{n} a_i(t)\Phi_i(x, v), \tag{3.1}$$

as descried in [5]. Temporal dependence rendered through $a_i(t)$ is independent from the spatial information carried in $\Phi_i(x, v)$. Here $\Phi_i(x, v)$ is called the i-th basis mode. With increasing i, the accuracy of the solution consequently increases as well, which is similar to increasing the spatial resolution in finite difference methods outlined in chapter 2. The essence of dimensionality reduction algorithms is now to find optimal basis modes $\Phi_i(x, v)$. Optimality here means capturing the dynamic answer to a given geometry and inital conditions thus permitting to exploit a minimal number p of basis modes to reconstruct the dynamics.
An optimal basis can be provided by POD. It leverages a "physically interpretable spatio-temporal decomposition"[5][p.375] of the input data . At first this data needs to be preprocessed in a fashion, that seperates the temporal and spatial axis. Each temporal state is called snapshot and is stacked in a matrix P such that

$$\mathrm{P} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n] \qquad \text{with} \qquad \mathbf{u}_i = [f(v_1, x_1), \ldots, f(v_k, x_1), f(v_1, x_2), \ldots, f(v_k, x_j)]^T, \tag{3.2}$$

where n is the number of available snapshots and $\mathbf{u}_i$ the i-th snapshot with $\mathbf{u}_i \in \mathbb{R}^{j \times k}$. Afterwards P is decomposed using the singular value decomposition (SVD) which solves the left and right

singular value problem leveraging

$$\mathrm{P} = U\Sigma V^*, \tag{3.3}$$

where U is a unitary matrix containing the left singular vectors of P in it's columns and V, also a unitary matrix, containing the right singular vectors in it's columns. Here superscribt asterix denotes the complex conjugate transpose. Furthermore Σ is a sparse matrix with the singular values in descending order on it's diagonal [5]. Note that the SVD always produces as much singular values and in turn singular vectors as there are elements on the shortest axis of the input matrix. Hence when preprocessing the input data as described above one gets as many singular values -and vectors as there are snapshots, given that the resolution in time is smaller than the spatial resolution.
Next, by applying the Eckard-Young theorem, that can be looked up in [5], it is possible to solely harness the first leading singular values and corresponding vectors to approximate P to a desired accuracy. The theorem states that the the optimal rank-r approximation to P, in a least-squares sense, is given by the rank-r SVD truncation $\tilde{\mathrm{P}}$:

$$\underset{\tilde{\mathrm{P}}, s.t. rank(\tilde{\mathrm{P}})=r}{\operatorname{argmin}} \quad \| \mathrm{P} - \tilde{\mathrm{P}} \|_F = \tilde{U}\tilde{\Sigma}\tilde{V}^*. \tag{3.4}$$

Here $\tilde{U}$ and $\tilde{V}$ denote the first r leading columns of $\mathbf{U}$ and $\mathbf{V}$, and $\tilde{\Sigma}$ contains the leading $r \times r$ sub-block of Σ. $\| \cdot \|_F$ is the Frobenius norm [5].
When decomposing a matrix that contains snapshots of a dynamical system, the columns of $\tilde{U}$ and $\tilde{V}$ contain dominant patterns that describe the dynamical system. Moreover they provide a hierarchical set of modes, that characterize the observed attractor on which we may project a low-dimensional dynamical system to obtain reduced order models. That being said, we use the left singular values of $\tilde{U}$ as optimal basis modes such that

$$\tilde{U} = \Phi = [\Phi_1, \Phi_2, \ldots, \Phi_r]\,. \tag{3.5}$$

Vectors in Φ are orthogonal to each other and in that way provide a coordinate transformation from the high dimensional input space onto the low dimensional pattern space.[PLS CHECK AGAIN, sentence is stolen from KUTZ]

3.2 Autoencoders

Another way to obtain an optimal basis is to employ a machine learning architecture situated in the field of deep learning called autoencoders. An autoencoder is a feed forward neural network, that is trained to learn salient features of it's input by compressing it and successively reconstructing it back from the compressed version. In that way one could say that an autoencoder simply copies it's input to it's output [7]. Figure 3.1 shows a schematic representation of the architecture for a deep undercomplete autoencoder and the necessary terminology used to describe it's components. For the moment we ignore the difference between an undercomplete and overcomplete autoencoder, and come back to that when talking about regularization.
The biggest allocation unit in autoencoders is the distinction of a decoder and a encoder part, which are seperated at central code layer. The encoder is a mapping h. The encoder maps the input P to the code Φ while compressing it, which is written as $h(\mathrm{P}) = \Phi$. Hence the encoder consists of the input layer, a variable number of hidden layers and the code layer, which is the left side of the schematic autoencoder in fig. 3.1. The decoder mirrors the encoder, which is not a necessity but

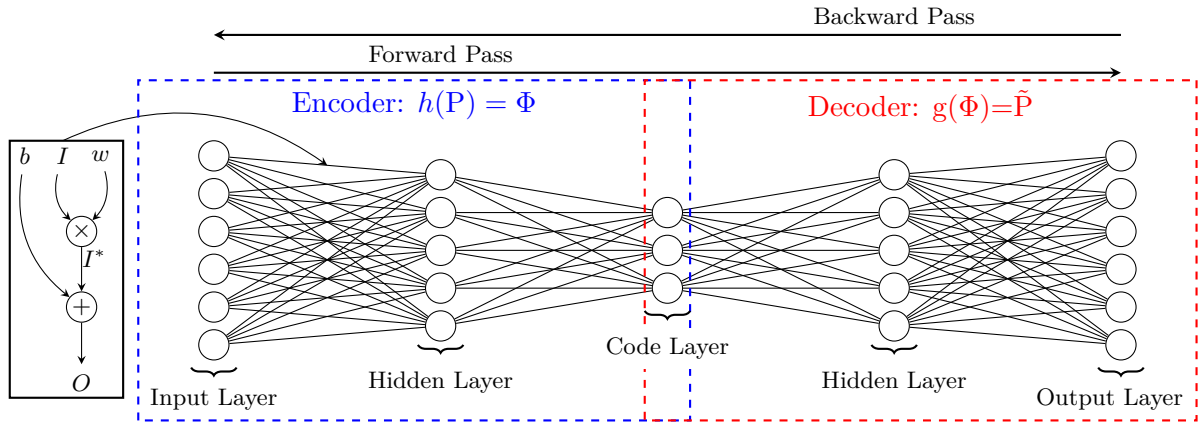


Figure 3.1: Scheme of an undercomplete autoencoder with five fully connected layers. Input layer, output layer, the code or bottleneck layer and two hidden layers. Every layer is made up of nodes, which are represented as circles. The more hidden layers are added to the architecture the deeper is the autoencoder and the greater the capacity of the model. Not shown are possible activations for each layer. Labelled is decoder and encoder. The decoder is a mapping $g(\Phi) = \tilde{\mathrm{P}}$, taking Φ as input and outputs an approximation $\tilde{\mathrm{P}}$ of P. Similarily the encoder is a mapping $h(\mathrm{P}) = \Phi$, taking Φ as input and outputs the code Φ. The box left of the autoencoder shows a computational graph for the feed forward connection of two nodes. The input of a node is I which is multiplied with w a weight. A bias b is added to the result I^* which yields the output O, input of a node right of the inital node.

often chosen that way, see the right side of fig. 3.1. It comprises the same code layer, a variable number of hidden layers and the output layer. Similarily it is a mapping g which maps Φ to $\tilde{\mathrm{P}}$, an approximation to the initial P, and is written as $g(\Phi) = \tilde{\mathrm{P}}$. The number of hidden layers in encoder and decoder is the first of the so called hyperparameters that can be tuned to improve performance of the autoencoder. Note that we refer to autoencoders with more than one hidden layer, the code layer, as deep autoencoders. The reverse is a shallow autoencoder. The second largest allocation unit in autoencoders (and in general neural networks) are the layers. Each layer can be vector, a matrix or a three dimensional tensor. For the time beeing we choose each layer to be vector valued in the following. Therefore the input layer can be an input vector like $\mathbf{u_1} \subseteq \mathrm{P}$. The vectors in the hidden layers are abstractions of the input. The term hidden stems from the fact, that one usually does not look at them as the interest mainly lies in the code -and output layer. The code layer is of main interest, as is holds the compressed abstraction of the input. Coming to the smallest allocation unit, the nodes. Each node refers to an entry in the vector/layer and is dispalyed as a circle in fig. 3.1. The number of nodes per hidden layer is a second hyperparameter. Two nodes are connected in a forward pass through solving $O = I * w + b$, which is represented as a computational graph in fig. 3.1. Here the input is I, which is multiplied with a weight w and by adding a bias b, one obtains the output O. Hence every connection between nodes contains two free parameters: a weight w and a bias b. Hence the whole network holds a set of parameters which we call $\theta \in \mathbb{R}$ in the following. A forward pass in this sense refers to the flow of information from the left side of the encoder to the right side of the decoder in fig. 3.1. All of the aforementioned can be found in [7].

3.2.1 Training

In a feed forward neural network, as the autoencoder used in this thesis, the information flows forward from input to output layer when we want to perform an evaluation as in $AE(\mathrm{P}) = g(h(\mathrm{P})) = \tilde{\mathrm{P}}$. In that way the layer structure in fig. 3.1 can be viewed as evaluating a composition of functions

$$l^{(4)}(l^{(3)}(l^{(2)}(l^{(1)}(l^{(0)}(\mathrm{P}))))) = g(h(\mathrm{P})) = AE(\mathrm{P}) = \tilde{\mathrm{P}}, \tag{3.6}$$

where every function represents one layer. Here $l_{(0)}$ and $l_{(4)}$ are input layer and output layer, $l_{(1)}$ and $l_{(3)}$ the hidden layers in encoder and decoder with $l_{(2)}$ the bottleneck layer. The evaluation of one layer or function is a linear transformation of the incoming data with

$$\tilde{\mathbf{u}}_1 = \mathbf{u}_1 W + \mathbf{b}. \tag{3.7}$$

Here $\mathbf{u}_1$ is as in eq. (3.2), W is the weight matrix and $\mathbf{b}$ a bias vector, which is the same equation as shown in the computational graph in fig. 3.1. The weight matrices and bias vectors of each layer are collected in the set $\theta \in \mathbb{R}$. It is self evident, that θ does not minimize the cost function $J(\theta)$ from the start with

$$J(\theta) := \frac{1}{n}\sum_{i=1}^{n}(\mathbf{u}_i - AE(\theta; \mathbf{u}_i))^2, \tag{3.8}$$

and needs to be found through a learning process, which is called training. The training will be discussed in the following.
Note that the cost function $J(\theta)$ comprises the mean squared error over all snapshots $\mathbf{u}_i$, which we will write as $L(\mathrm{P}, \tilde{\mathrm{P}})$ the so called loss with

$$L(\mathrm{P}, \tilde{\mathrm{P}}) := \frac{1}{n}\sum_{i=1}^{n}(\mathbf{u}_i - \tilde{\mathbf{u}}_i)^2 = E, \tag{3.9}$$

when specifically referring to E the distance between P and $\tilde{\mathrm{P}}$. In order to minimize $J(\theta)$ we compute the gradient of $J(\theta)$ with respect to the free parameters θ written as $\nabla_\theta J$. Specifically we need to compute the derivative of the network with respect to the free parameters as seen in eq. (3.8). Then we move in the opposite direction of the gradient, which yields an update of θ as

$$\theta \leftarrow \theta - \epsilon\, \mathbf{g}\,, \qquad \text{where} \qquad \mathbf{g} := \nabla_\theta J. \tag{3.10}$$

Backpropagation: The computation of the gradient $\mathbf{g}$ is done in the so called backpropagation. As the name implies the information flows backwards through the network, from output to input layer, and propagates E backwards through the network. Responsible for this backward motion is the recursive application of the chain rule of calculus. Considering as a simple illustrative example fig. 3.2 with $y = g(z, b) = g(f(x, a))$, a single node connection from input x to output y with a single hidden node. Then a is a weighting constant of the first node and b defines a weighting consant of the hidden node. Backpropagating the distance E with $E = \frac{1}{2}(y_0 - y)^2$ between the output and the ground truth y_0 though the network yields

Figure 3.2: A single node network with one hidden node between input and output node.

$$\frac{\partial E}{\partial a} = -(y_0 - y)\frac{\partial y}{\partial z}\frac{\partial z}{\partial a} \qquad \text{and} \tag{3.11}$$

$$\frac{\partial E}{\partial b} = -(y_0 - y)\frac{\partial y}{\partial b}, \tag{3.12}$$

which requires $\frac{\partial E}{\partial a} = -(y_0 - y)\frac{\partial y}{\partial z}\frac{\partial z}{\partial a} = 0$ to minimize E. The update rule is then given by

$$a_{i+1} = a_i + \epsilon\frac{\partial E}{\partial a_i} \qquad \text{and} \qquad b_{i+1} = b_i + \epsilon\frac{\partial E}{\partial b_i}. \tag{3.13}$$

This example is taken from [5] and shows the manner in which the chain rule of calculus provides the backwards direction when deriving the gradients of each layer with respect to the free parameters.

Generalization: At this point the objective for training a neural network is specified. The difference between pure optimization and learning lies in their differing objectives. In pure optimization a cost function like $J(\theta)$ is minimized in order to fit P to $\tilde{\mathrm{P}}$. This is called the objective of the optimization task. Learning on the other hand uses the same objective as the optimization task, but equally cares about the so called generalization task. In machine learning generalization refers to the ability of the learning algorithm to not only fit the input data, but also to fit data it has not seen before. It does so by using salient features of the input data which enables generalization. This can only be achieved to a certain extend and is sometimes intractable. But how is the generalizing performance measured? We randomly shuffle the input data P and split it into a training and a validation set in a 80/20 fashion with $\mathrm{P} = \{\mathrm{P}_{train}, \mathrm{P}_{val}\}$. The random shuffling is performed as to eliminate any bias on both sets. Then we minimize $J(\theta)$ using only P_{train} which minimizes $L(\mathrm{P}_{train}, \tilde{\mathrm{P}}_{train})$ and check if indirectly $L(\mathrm{P}_{val}, \tilde{\mathrm{P}}_{val})$ is beeing minimized as well. The indirect minimization of $L(\mathrm{P}_{val}, \tilde{\mathrm{P}}_{val})$ is called the generalization task. The loss over the validation set is called the validation error and the loss over the training set is called the train error. Note that in learning both tasks are equally valued. As mentioned before, we want to learn an optimal basis Φ which specifically contains salient features of P. With minimizing only the optimization task, we acquire the same basis as in POD. With learning we try to find an even more powerful basis Φ that enables the network to generalize.

Adam: The algorithm to compute the updates to θ is called Adam, an upgraded version of the classic stochastic gradient descent algorithm (SGD) with moments. The name stems from adaptive moment estimation, which refers to the adaptation of moments which in combination with the learning rate ϵ apply scaled, directional updates to θ during training. Moments and how they are adapted are discussed a bit further on. The steps of Adam are shown in algorithm 1 and explained successively. There we initially define hyperparameters which are the step size/learning rate ϵ, exponential decay rates d_1 and d_2 in $[0, 1)$ and a small constant δ for numerical stability. We can use the default values for all constants with $\epsilon = 1e-2$, $d_1 = 0.9$, $d_2 = 0.999$ and $\delta = 1e-8$. Despite the learning rate ϵ, Adam is fairly robust w.r.t. changing it's hyperparameters. Hence solely the hyperparameter ϵ requires tuning. Next we initialize the 1st and 2nd moments $\mathbf{s}$ and $\mathbf{r}$ to zero, in contrast to θ, which is initialized as follows.

The selection of starting values for θ is of crucial importance for the success of any neural network to learn something useful in a reasonable time. Moreover it strongly affects if the learning algorithm converges at all. This is partly because with the starting point we introduce a strong bias on the network considering that the SGD algorithm only makes marginal contributions to θ at every update. Even if $J(\theta)$ reaches a minimum, different initial values can lead to a variety of generalization errors. Specifically this connection between initial parameters and generalization is not well understood. Note that the topic of parameter initialization is despite it's importance, only sketched out briefly in this thesis. For more details see [7] for an overview, [17] for the default initialization in `pytorch`, which is also used in this thesis and [18] for very deep networks with more than eight layers and rectifying non-linear activations.

Usually the biases can be chosen to a heuristically constant value, while the weights need to be initialized so that they break symmetry between nodes. Imagine two hidden nodes with the same activation function connected to the same input. The same gradient would lead to a symmetric evolution during training, making one of the nodes redundant. Hence weights are initialized randomly often drawn from a Gaussian distribution. Next, finding the right scale for the distribution, which in turn scales the initial weights, is tackled. The spectrum of initial weight sizes ranges from large initial weights which are beneficial for breaking any symmetry, to small initial weights or even a sparse initialization. Now with this in mind, if we scale the distribution to produce large weights, we induce a prior, which states that all nodes are connected and how they are connected. Small initial weights on the other hand induce a prior, that says that it is more likely for nodes to not be connected and that the learning algorithm can choose which nodes to connect and how strong.
There exist several heuristics for the scale of the initial weights, which try to preserve the norm of the weigh matrix of each layer, to stay close to or below unity. Envisage that during forward as well as backward propagation we perform matrix multiplications from one layer to the next. Keeping the norm of each layer from exceeding unity, we prevent the development of exploding values and gradients in the respective last layer. A preventive measure is to scale the distribution for one layer with respect to it's non-linear activation with a factor called gain. Gains for non-linear activations can be found in [19] along with several sampling methods. The effects described above become severe with large layers or equally with networks of a certain depth, which is not the case for the autoencoders used in this thesis allowing to comfortably choose the default initialization in `pytorch` for linear -and convolutional layers where we draw the initial weights from a normal distribution with $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$ as suggested in [17]. The number of input nodes, also called fan-in, per layer is m.

Algorithm 1: The Adam algorithm

Require: Step size ϵ
Require: Exponential decay rates from moment estimates d_1 and d_2 in $[0, 1)$
(Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerial stabilization (Suggested default: 10^{-8})
Require: Initial parameters θ
Initialize 1st and 2nd moment variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$;
Initialize time step i = 0;
while *stopping criterion not met* **do**
Sample a minibatch $\mathrm{P}_{\mathrm{t}i}$ containing m examples from the training set P_{train};
Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_m L(\mathrm{P}_{\mathrm{t}i} - AE(\theta; \mathrm{P}_{\mathrm{t}i})$;
$t \leftarrow t + 1$;
Update biased first moment estimate: $\mathbf{s} \leftarrow d_1\mathbf{s} + (1 - d_1)\,\mathbf{g}$;
Update biased second moment estimate: $\mathbf{r} \leftarrow d_2\mathbf{r} + (1 - d_2)\,\mathbf{g} \circ \mathbf{g}$;
Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-d_1^i}$;
Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-d_2^i}$;
Compute update: $\Delta\,\theta = -\epsilon\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$ (operations applied element-wise);
Apply update: $\theta \leftarrow \theta + \Delta\,\theta$
end

SGD is derived from the common gradient decent method, where the whole batch P_{train} is used per iteration to compute the gradient for an update of θ in the opposite direction of the gradient. Hence gradient descent is a so called batch gradient method or deterministic gradient method. SGD is a stochastic method. It approaches updates for θ as in eq. (3.10) in a stochastic manner, using only small portions of P_{train} per update. By randomly sampling only a small number of examples from P_{train} and taking the average gradient over those examples we compute an estimate of the true gradient which is much cheaper. Therefore SGD trains with so called minibatches of a certain size, which yields $\mathrm{P}_{train} = \{\mathrm{P}_{t_i}, \ldots, \mathrm{P}_{t_n}\}$. Here i counts through all the minibatches and n is the total number of minibatches. The size of one minibatch can be obtained through dividing the dimension of your input, on which you split the batch, by n and is called κ in the following and is another hyperparameter. Minibatch sizes κ are influenced by the following observations. As one minibatch can be processed in parallel to compute an update, extremely small κ underutilize multicore architectures. Extremely large κ on the other hand can be limited by available memory to be solved in parallel. When utilizing a graphics processing unit (GPU) κ of the power of two is advised to fully exhaust the whole hardware (array sizes of the power of two achieve better runtime in GPUs). Small κ can have a regularizing effect hence improve generalization, but also amplify the noisy behavior [7][p.270ff].
Stochasticity is key in SDG and is achieved by shuffling the input data, which was already performed for the training and validation split. On every update the gradient is evaluated as an average over a minibatch of randomly sampled examples, yielding an unbiased estimate of the true gradient. This measure leads to noisy updates of θ which improves generalization and converges faster than common gradient descent. SGD has the ability to find a minimum even before the whole batch is processed, especially with large datasets, as elaborated in [7][p.286 ff]. The noisy updates require a small learning rate, especially when the the minibatch size κ is small, that decays over time as the noise does not vanish when a minimum is found. Up to now we saw that small minibatches improve generalization, underutilize available multicore hardware and amplify noisiness which needs to be tackled with small learning rates. Therefore the learning process is slowed down. Here momentum comes into play, which accelerates learning.
Momentum applies an exponentially decaying moving average of previous gradients to the current gradient. Momentum is an analogy to momentum in physics and therefore named after. Imagine a tightrope walker, who uses the moment of inertia of a long rod for balance as crossing the rope. Oscillations of the walker need to overcome the moment of inertia of the long rod in order to destabilize the walker. Another analogy is a fast car taking a sharp turn. The car drifts because the new direction is influenced by the old orthogonal momentum. If we take the update rule for the first moment estimate $\mathbf{s}$ of algorithm 1 and look at three consecutive updates

$$\mathbf{s}_i = d_1\mathbf{s}_{i-1} + (1-d_1)\mathbf{g}_i, \tag{3.14}$$
$$\mathbf{s}_{i-1} = d_1\mathbf{s}_{i-2} + (1-d_1)\mathbf{g}_{i-2}, \tag{3.15}$$
$$\mathbf{s}_{i-2} = d_1\mathbf{s}_{i-3} + (1-d_1)\mathbf{g}_{i-3}, \tag{3.16}$$

we see that by combining and simplifying the three updates yields

$$\mathbf{s}_i = d_1 d_1(1-d_1)\mathbf{g}_{i-2} + \cdots + d_1(1-d_1)\mathbf{g}_{i-1} + \cdots + (1-d_1)\mathbf{g}. \tag{3.17}$$

Note that this entanglement of the updates is taken from [20]. Obviously the contributions to the current gradient vanish exponentially over iterations. This leads to a moving average of past

and current gradients. Now taking into account that if eq. (3.14) computes the moment of the first iteration, there are no previous gradients available which leads to a biased first computation. Hence we divide $\mathbf{s}_i$ by $(1 - d_1^i)$ for the bias correction of the first moment in algorithm 1 which yields $\hat{\mathbf{s}}$. In consecutive iterations obviously the bias correction approaches zero. All the same steps does the second biased moment estimate $\mathbf{r}$ take. The second moment estimate is nothing more than the squared past gradients gradients written as $\mathbf{g} \circ \mathbf{g}$ in algorithm 1. Together they from the parameter update $\Delta\,\theta$. Without going into further details, it can be shown that the parameter updates have an approximate upper bound of $\Delta\,\theta \lessapprox \epsilon$, see [21]. This upper bound is a necessary criteria for the updates, as the learning rate ϵ is a specifically tuned hyperparameter that should not be exceeded during training. The ratio $\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$ is called signal to noise ratio (SNR) in [21], which evolves towards zero as an optimum is reached. Furthermore, the updates are invariant to the scale of the gradients as a factor a is always canceled out with $(a \cdot \hat{\mathbf{s}})/\sqrt{(a^2 \cdot \hat{\mathbf{r}})} = \hat{\mathbf{s}}/\sqrt{\hat{\mathbf{r}}}$. To sum up, using first and second momentum we can accelerate training as the moving average gradient smooths the noisy updates and in turn is a better estimate of the true gradient, the bias in the first iteration is corrected and we stay invariant to gradient scaling.
The flowchart in fig. 3.3 shows how the generalization query, weight initialization, backpropagation and the integration of Adam into the training procedure is carried out within this thesis. The network is trained in two loops, where the outer runs over so called epochs and the inner over all minibatches. One epoch is completed after all minibatches are shown to the network and the validation error is calculated as well as the train error, for which we take the average over all minibatches. The number of epochs H is determined based on over- and underfitting as well as on how many epochs are needed for Adam to converge and is another hyperparameter. In the following capacity, regularization, over- and underfitting is described.
Capacity, regularization, over- and underfitting: In order to satisfy the objective of learning an optimal basis in the code layer, autoencoders need to be regularized. Regularization is defined in [7] as any modification made to a learning algorithm that is intended to reduce its generalization error but not it's training error. As already mentioned autoencoders have a central hidden layer between the equally sized input -and output layer. In undercomplete autoencoders the central hidden layer is also called bottleneck layer, because it's size is smaller than the input. Thus the undercomplete autoencoder is forced to decide which information to copy and by that measure is regularized. Overcomplete autoencoders on the other hand have the central hidden layer greater or of the same size as the input, hence making it indispensable to use additional regularization, which, necessary to add, can be also fruitful in undercomplete autoencoders. Note that an undercomplete autoencoder as shown in fig. 3.1 is used in this thesis which is called autoencoder for simplicity. Regularization is always advantageous when the capacity of an autoencoder is too big. The capacity of any neural network can be altered through the number of free parameters available for the model to fit the input data. Recall that an autoencoder has a variable number of hidden layers (excluding the bottleneck layer) of variable size and thus a variable number of free parameters. Imagine having a model big enough in terms of free parameters to memorize the whole training set, we would achieve good results for the optimization task, but miss out on learning useful features thus failing at the generalization task. This relationship is exemplary shown in fig. 3.4 where a point of optimal capacity is defined. Optimal capacity is reached when optimization and generalization in the form of training error and validation error are in balance. Note that fig. 3.4 is taken from [7][p.112]. Another way that alters the capacity of a neural network is adding non-linear activation functions to layers. So far we saw that every node and by that layers are connected through linear

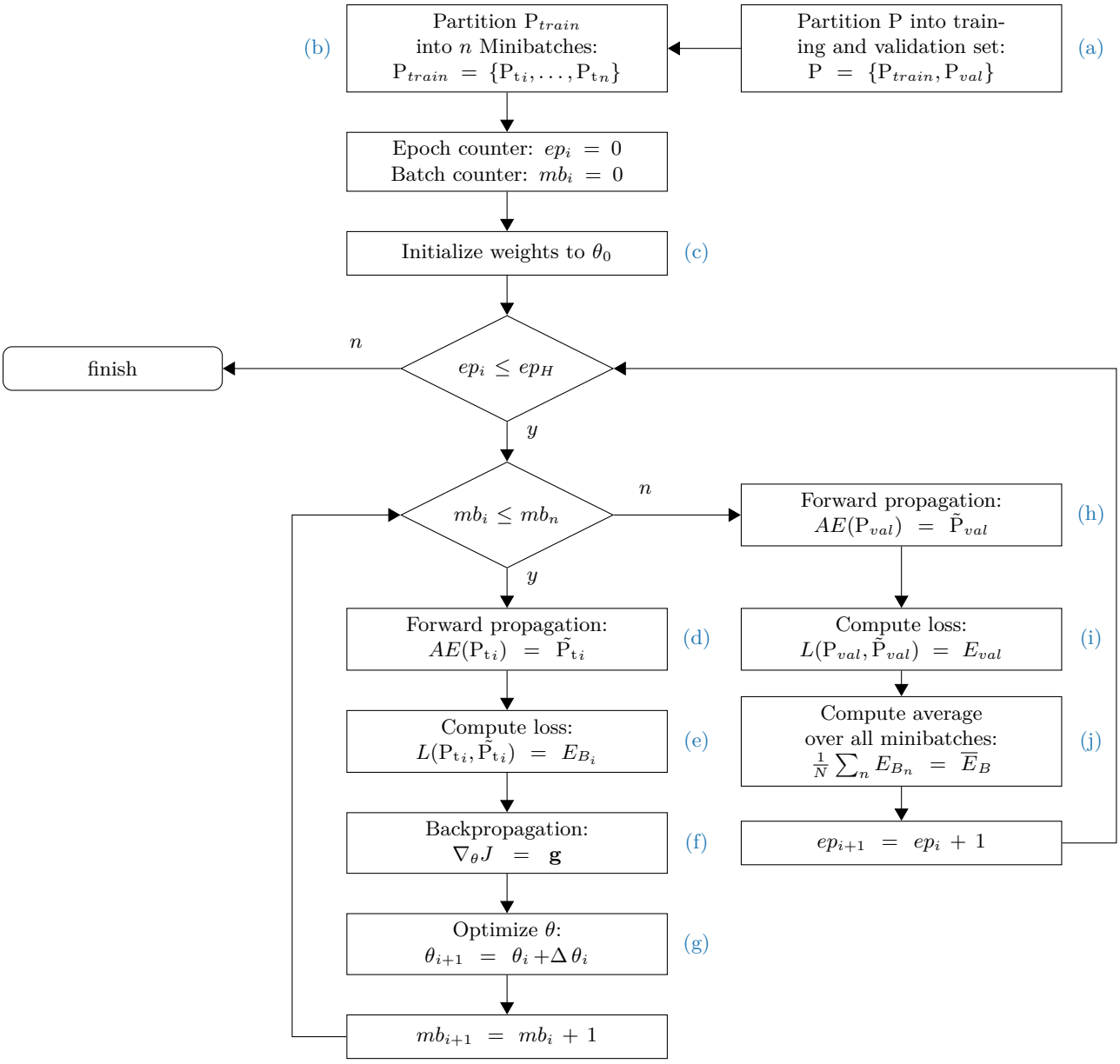


Figure 3.3: Flowchart of the training process used in this thesis for an autoencoder. The data set P for training is split up into a training P_{train} and a validation P_{val} set with a 80/20 ratio (a). The P_{train} set is equally devided into n minibatches (b). The network will be trained over H epochs and n minibatches. First the weights w are initialized as in [17] and biases set to zero (c). A successive evaluation of the first minibatch with $AE(P_{t_i}) = \tilde{P}_{t_i}$ called forward propagation takes place in (d). The mean squared error between P_{t_i} and $\tilde{P}_{t_i}$ yields the average error over one minibatch E_{B_i}. Thereafter E_{B_i} is backpropagated through the network (f) yielding the gradient $\mathbf{g}$. This gradient is then used to optimize weights w and biases b with *Adam*, an optimization algorithm seen in algorithm 1, (g). In the fashion of using (d) to (g) all n minibatches are shown to the network and lead to an optimization of the network's free parameters θ which are weights w and biases b. After all minibatches are shown to the network a validation step is taken. Hence the network evaluates P_{val}, which it has not seen before, in (h) with $AE(P_{val}) = \tilde{P}_{val}$. Subsequently, the evaluation of the mean squared error between P_{val} and $\tilde{P}_{val}$, produces the validation error E_{val} in (i). Afterwards the arithemtric mean of all minibatch errors E_{B_i} is taken. This produces $\overline{E}_B$ and concludes the first epoch. The maximum number of epochs H is reached when E_B and E_{val}, have dropped to a satisfactory value and the training finishes.

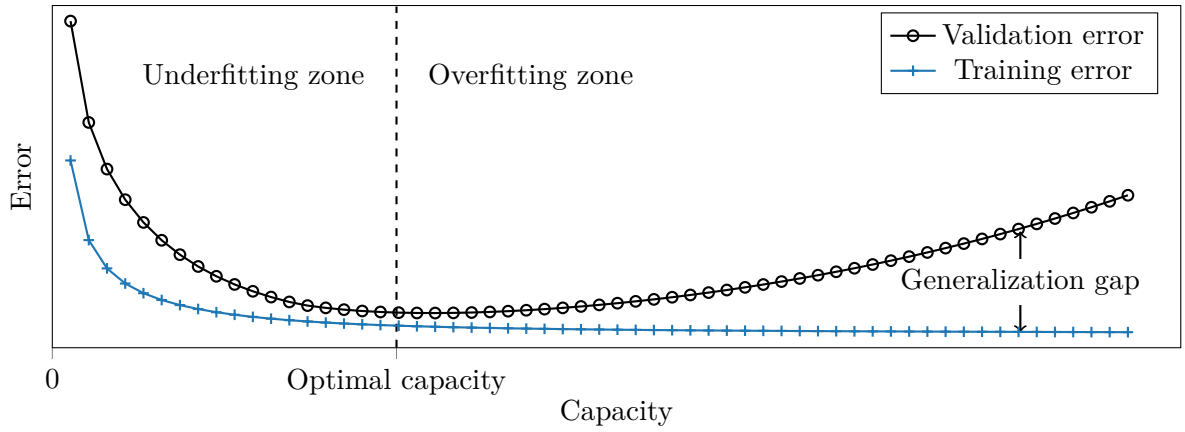


Figure 3.4: Figurative example of how capacity influences the evolution of training and validation error. With increasing capacity the model i.e. autoencoder is able to fit the training and validation set thus training and validation error decreases. Typically the training error is less than the validation error. Yet both errors are too high and the model is underfitting. Further increasing the capacity leads to an increase of the validation error while the training error further decreases. The gap between training- and validation error is called the generalization gap. Once the generalization gap dominates the decrease in training error the model is overfitting and capacity passed the point of optimal capacity. Optimal capacity is the point where both optimization and generalization are in balance.

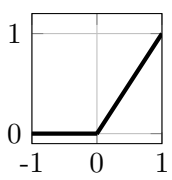
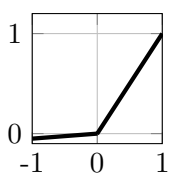
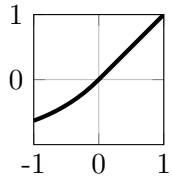
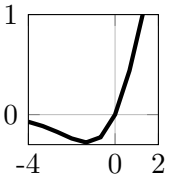
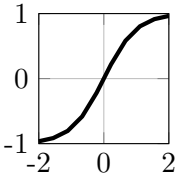
transformations as in eq. (3.7). Thus the networks so called hypothesis space solely includes linear transformations of the input. The hypothesis space from which a neural network can choose the composition of the solution can be enriched with non-linear functions by simply evaluating layers through non-linear functions σ, also called activations, leading to

$$\tilde{\mathbf{u}}_1 = \sigma(\mathbf{u}_1 W + \mathbf{b}). \tag{3.18}$$

Activations: Non-linar activations typically used for neural networks are summarized in table 3.1. In this thesis these are also the activations used in the final network design or during hyperparameter search. We use two classes of activations. One is of type rectifier, which are linear for positive inputs and are zero, or decay below linear for negative inputs. The other is the tangens hyperbolicus (Tanh) and the Sigmoid-weighted linear unit (SiLU). Tanh and sigmoid share a similar "S"-shaped curve hence are put in one class. SiLU can be written as SiLU $= \frac{1}{2}x(1 + \tanh(\frac{x}{2}))$. Note that Tanh saturates for input values exceeding the range $[-2, 2]$ which leads to a so called vanishing gradient. In the most severe case all updates can be zero. However in this thesis a vanishing gradient was not encountered. Theoretically it is possible to take any function as an activation. Though indispensable is their differentiability, which reduces to partly differentiable for rectifiers, to allow backpropagation. Activations σ and where in the network to place them is yet another hyperparameter.

Layer types: So far we have adoped the idea that, for two successive layers, all nodes from the first layer are connected to all nodes from the following layer and that all layers are vectors. Hence the types of layers are called fully connected or `Linear` in `pytorch`[19]. If we now want the input to be a vector, a matrix or a three dimensional tensor we have to adopt another type of layer called convolutional layer. In `pytorch` these layers are called `Conv1D`, `Conv2D` and `Conv3D` respectively. Let us start with a simple example illustrating conceptual features of convolutional layers. A digital image can be viewed as a distribution of pixels over a two dimensional surface, where the surfaces dimensions are described with the images width and height. Furthermore, each pixel has

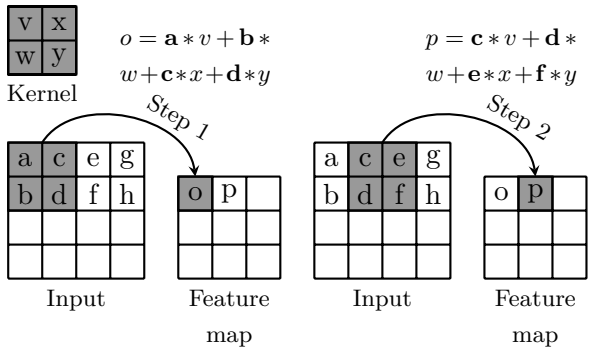
Table 3.1: Non-linear activation functions of type rectifier are the rectified linear unit (ReLU), it's leaky variant LeakyReLU, the exponential version ELU. The negaive slope of LeakyReLU below zero α is typically, and also in this thesis set to $\alpha = 0.001$. Activations based on tangens hyperbolicus (Tanh) are the sigmoid function with $\text{sig}(x) = \frac{1}{2}(1 + \tanh(\frac{x}{2}))$ and it's variant SiLU. The functions are shown over an illustrative domain.

ReLU	LeakyReLU	ELU	SiLU	Tanh
$\sigma = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$	$\sigma = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$	$\sigma = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$	$\sigma = x\text{sig}(x)$	$\sigma = \tanh(x)$
				

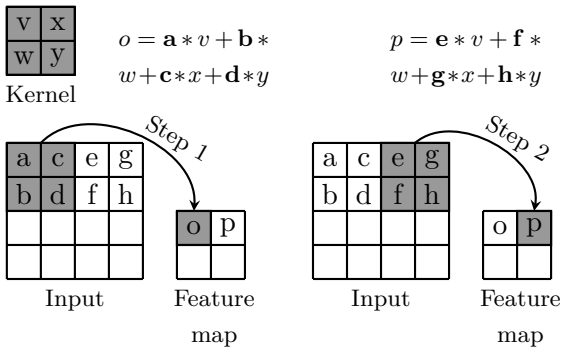
a color expressed in values of the primary colours red, green and blue (RGB). Hence each pixel is a combination of different RGB values. The RGB values for each pixel are stored in so called channels. The red channel, green channel and blue channel. This concludes, that the dimension of a digital image is additionally equipped with the channel dimension. Let an image be Img, then $Img \in \mathbb{R}^{m \times n \times c}$, with m the width this image, n the height of this image and c the three RGB channels. Another important characteristic of images is the spatial correlation between neighboring pixels. This can be for example sharp corners that separate foreground from background or differentiate between entities portrayed in the image. Therefore similarly as saving information about colour composition in channels, we can imagine saving spatial information in additional channels. For example corners and other geometric shapes saved in a channels for themselves. With this idea in mind convolutional layers can be introduced. Convolutional layers comprise a so called kernel, which moves over the image and by that all the input channels. Therefore, if the kernel is K then $K \in \mathbb{R}^{i \times j \times c}$. The height i and width j of the kernel can be adjusted and is usually smaller than the input's height and width. The c dimension is always equal to the c dimension of the input. The distance the kernel travels over the input at every step is specified with the so called stride s. For a two dimensional input image like Img the strides are given in direction s_1 and s_2, where the latter corresponds to m and the former corresponds to n of Img. The kernel performs the same linear transformations of the input as fully connected layers at every part of the input visited. This action outputs a so called feature map, which as the name implies should capture features of the input. A comparison of a one strided convolution with $s_1 = s_2 = 1$ and a two strided convolution with $s_1 = s_2 = 2$ is given in fig. 3.5. The one strided convolution yields a feature map with a reduction of m and n by one, while the two strided convolution yields a feature map with a reduction of m and n by 2. Note that in fig. 3.5 eventual channels are omitted.
Even tough the name convolutional layer implies the use of the convolutional operation `PyTorch` and many other neural network libraries instead use the cross correlation operation, which is an operation closely related to a convolution [7] [**?**]. Details on why the cross correlation is used instead of the convolution is discussed further in [7]. In eq. (3.19) the two dimensional discrete cross correlation without channels is given with

$$M_{m,n} = \sum_{s_1,s_2,i,j} Img_{(m \times s_1)+i,(n \times s_2)+j} K_{i,j}. \tag{3.19}$$

There $M_{m,n}$ represents a point on a feature map, where m and n refer to the location of that point. The input to the cross correlation is exemplary our image Img where coordinates of a pixel are also given in m and n. The kernel is given as K with coordinates in i and j. The summation in eq. (3.19) is evaluated over i, j and s_1 and s_2. Imagine beeing on Img at $m = 1$ and $n = 1$ with $s_1 = 1$ and $s_2 = 1$, then the summation is performed over all pixels of Img that the kernel covers in i and j from the starting point. If we then evolve to $s_1 = 2$ and $s_2 = 1$ we can see that we start with the same summation at $m = 2$ and $n = 1$ as seen in fig. 3.5 (a). Hence the kernel K moved along the m axis of the input image Img. Therefore by evolving in s_1 and s_2 the kernel moves over the Img and eventually covers the whole input.



(a) Two dimensional example of of a **one** strided convolution over a 4×4 input matrix with a 2×2 kernel matrix. The equations for obtaining the components i and j of the feature map are given. Note that all biases are omitted for simplicity and bold symbols should help readability and do not represent vectors. The resulting feature map is a 3×3 matrix, as we downsampled the input by a factor of 0.75.



(b) Two dimensional example of of a **two** strided convolution over a 4×4 input matrix with a 2×2 kernel matrix. The equations for obtaining the components i and j of the feature map are given. Note that all biases are omitted for simplicity and bold symbols should help readability and do not represent vectors.. The resulting feature map is a 2×2 matrix, as we downsampled the input by a factor of 0.5.

Figure 3.5: Comparison of a one strided convolution (a) and a two strided convolution (b). Illustrated are two steps of a kernel matrix moving over an input matrix. The two strided convolution yields an increased downscaling compared to the one strided convolution.

Next, if we want more feature maps, because we may assume many features are present in the input, we can specify more than one kernel to move over the input, each yielding a different feature map. In fig. 3.6 an illustrative example of a convolutional neural network with four convolutional layers, which downsamples the width and height of the input in every layer while at the same time adds channels (feature maps) in every layer. When arriving at the last convolutional layer, the featuremaps are flattened and fully connected to the code layer. A comparable scheme for the encoder side of a convolutional autoencoder is used in this thesis. When the encoder in fig. 3.6 is mirrored at the code layer and additionally all convolutional layers are replaced with transposed convolutional layers, we get the decoder and by that arrive at the typical autoencoder architecture as in fig. 3.1. In conclusion, up to now we have identified four more hyperparameters for a single convolutional layer: kernel width and height, number of kernels i.e. feature maps, which are usually called output channels and stride.
Fully connected layers are able to down- and upsample making them applicable for the decoder and encoder in autoencoders. Convolutional layers on the other hand are not able to upsample a given input. Therefore the transpose of a convolutional layer is applied for upsampling. The re-

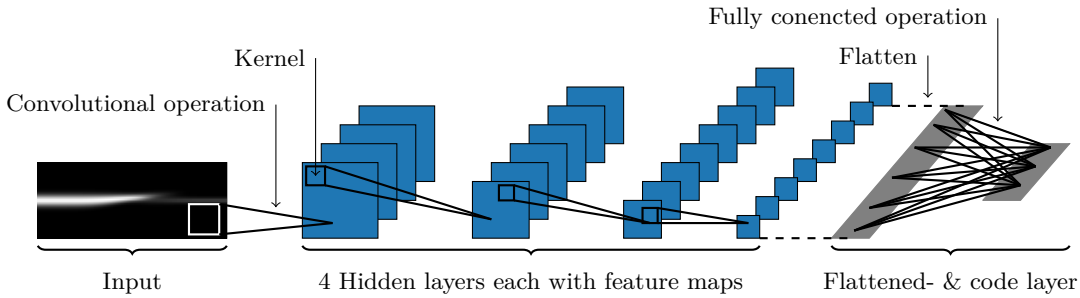


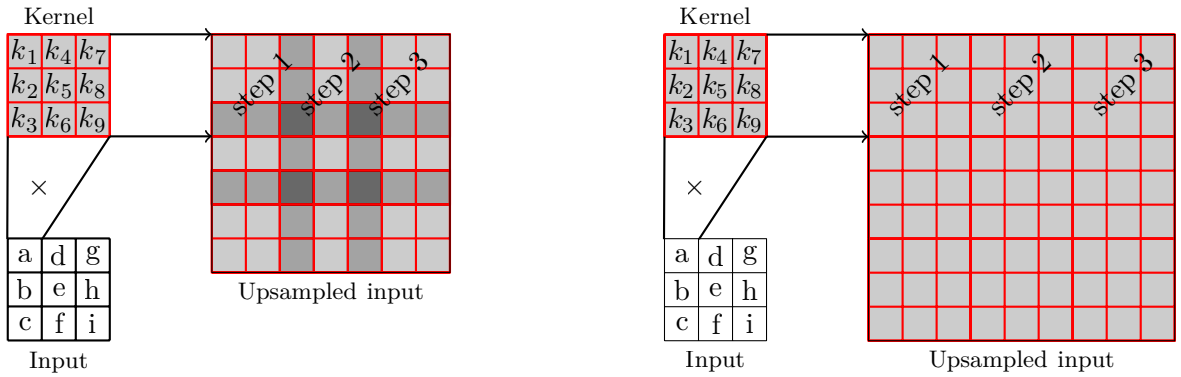
Figure 3.6: Schematic representation of typical components and their structural set-up in a convolutional neural network (CNN). Shown is the convolutional operation and the associated kernel, the flattening and successive fully connected operation. First a kernel moves over the one dimensional input performing convolutional operations, which yields the components of a feature map in the successive hidden layer. One channel in a hidden layer can aswell be called feature map. Here the input has just one channel, but four different kernels move over the input which produces four feature maps in the first hidden layer. Hence the first hidden layer comprises four channels or feature maps. Whenever one layer has different channels, the kernel moves with different parameters over all channels at the same time. Therefore we can think of it in this case as a three dimensional kernel tensor. Note that for simplicity this is not depicted in this figure. While the width and height of the input decreases over the hidden layers, the number of channels typically increases . The last hidden layer is flattend and successively connected to the code layer through a fully connected operation.

spective modules for vectors, matrices and three dimensional tensors are called `ConvTranspose1d`, `ConvTranspose2d` and `ConvTranspose3d` in `pyTroch`. The transpose of a convolutional layer is not a transpose in the mathematical sense when tranposing a matrix for example. The transpose revers to recovering the previous shape. Furthermore they can be viewed as convolutins in which the backward- and forward pass are swapped, hence are often implemented that way [7] [22] [23]. A simplified sketch of a two dimensional transposed convolution without channels can be drawn by taking a point of the input and performing a linear transformation of that input with all kernel entries. The result is then the first part of the upsampled output located at the upper left corner. A successive point is taken and again linear transformed with all kernel entries to create the second part of the output, which is part of the upsampled output by s_1 strides counted from the upper left corner, away from the first part. This action can lead to overlapping entries on the ouput which are simply added. In fig. 3.7a this process is shown for an 3×3 input matrix, which is upsampled by a 3×3 kernel moving with stride length $s_1 = s_2 = 2$. The upsampled result is of size 7×7. With a kernel size of 3 in one direction and a stride $s_1 = 2$ in the same direction, there is always an unevenly distributed overlap in the output entries resulting in the checkerboard structure as seen in fig. 3.7a. This is a common problem with transposed convolutional layers. When the kernel width and height is divisible by the respective stride length s_1 and s_2 this issue can be mitigated, as the overlaps are evenly distributed. When the stride lengths $s_1 = s_2 = 3$ are for example of the same size as the kernel width and height no overlap occurs which is depicted in fig. 3.7b.
Obviously transposed convolutional layers comprise the same hyperparameters as convolutional layers which greatly influence the down- and upsample rate. But customizing both hyperparameters only for the donwsampling/upsampling rate can be detrimental for learning off the the input. Imagine having a kernel small enough to not be able to recognize overkill edges in pictures. Or similarly a step size large enough to overpass thin edges. Hence kernel size and stride width are somewhat determined by the structure of the input. By zero padding outlines of the input down- and upsampling rate can be tuned independently from kernel size and stride width. Zero padding

is available for convolutional layers as well as for the transpose of convolutional layers. It refers to padding the outer edges of the input with zeros. Putting everything together we can compute the size of the output for convolutional layers and transposed convolutional layers with

$$o = \left[\frac{i + 2p - k}{s}\right] + 1 \qquad (3.20) \qquad \text{and} \qquad o = (i-1)s - 2p + k \qquad (3.21)$$

respectively. Here all variables are linked to one dimension only like the width of an image for example. Hence o refers to the output size in one dimension, i to the input size in one dimension, $2p$ to the padding size for both edges in one direction, k refers to the kernel size in one dimension and s to the stride size in one direction. Note, whenever the stride width exceeds the input size in a particular direction, then the input is automatically padded with a necessary number of zeros on the left or bottom. Additionally nearly every transpose of a convolutional layer can be described by an associated convolutional layer for which the input is spread using zeros in between every point. More information and a detailed description of calculating output sizes of convolutional layers and transposed convolutional layers, which is named convolutional arithmetric by Dumoulin et. al, can be found in [24].



(a) Mechanism of a two dimensional **two** strided transposed convolution over a 3×3 input matrix with a 3×3 kernel matrix. Upsampled by a factor of 2.33 produces a 7×7 matrix. Three kernel locations on the output are indicated with step 1, step 2 and step 3. The extent of the kernel is not divisible by the stride width. Uneven overlapping procude a checkerboard like structure.

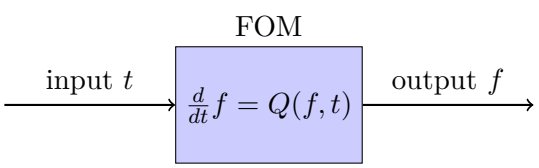
(b) Mechanism of a two dimensional **three** strided transposed convolution over a 3×3 input matrix with a 3×3 kernel matrix. The resulting output upsampled by a factor of 3 is a 9×9 matrix. Three kernel locations on the output are indicated with step 1, step 2 and step 3. The extent of the kernel is identical to the stride width. Overlaps and checkerboard like structures are avoided.

Figure 3.7: Visualization of the mechanism in transposed convolutional layers. Shown is pixelation, a common problem with transposed convolutional layers (a) and a possible solution, mitigating the pixelation effect.

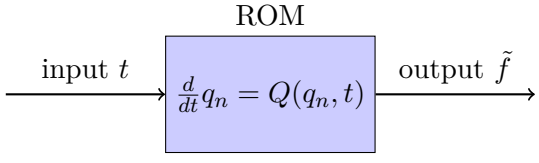
Neural netwroks and hence also autoencoders have many hyperparameters determining their capability for learning which were introduced in this section. There is little systematic knowledge about how the hyperparameters interact in the model. Goodfellow et al. point out that with a combination of intuition, certain methods and first of all experience practitioners find hyperparameters that work well [7]. As for this thesis a detailed description of the hyperparameter search for a fully connected- and convolutional autoencoder, in the following refferred to as FCNN and CNN, is provided in appendix A and appendix B respectively.

4 Model Order Reduction

In this chapter model order reduction (MOR) of the BGK-model in the Sod shock tube will be introduced for which POD and in particular autoencoders are adopted to obtain a reduced basis (RB).
Model order reduction is a technique used for reducing the computational cost when evaluating PDE's [6][2][25]. To achieve this, the solution to a PDE is being approximated by reducing one or more of it's dimensions. The reduction is performed by a mapping onto a low dimensional manifold. In our case the solution to the BGK-model is a function $f(x, v, t) \in \mathbb{R}^3$. Now we could reduce i.e. v to n, where k represents the number of elements in v and p represents the number of elements in n. With $p << k$ we obtain a reduced order model (ROM) which we call $q(x, n, t)$ of significantly lower dimension. In particular n is called the reduced basis or the intrinsic variables. In chapter 2 we saw that the BGK-model is a PDE which through discretization in the spatial dimension x and the velocity dimension v holds a system of KJ ODE's in time in 1D and K^3J^3 ODE's in time in 3D. By the reduction of v we arrive at nJ ODE's in time for 1D and nJ^3 ODE's in time in 3D. This example illustrates the amount of computations that can be saved by MOR. The mapping from $f(x, v, t)$ to $q(x, n, t)$ can be performed by one of the reduction algorithms from chapter 3. The remapping back to $\tilde{f}(x, v, t)$ is carried out by the same algorithm under the condition, that the distance $||f - \tilde{f}||$ is small.
To sum up, the idea that every high dimensional dynamical-state space W, also called solution manifold, where in our case $f(x, v, t) \in W$, can be mapped onto a state-space i.e. V of lower dimension with $q(n, \mu_i) \in V$, is exploited within MOR [25]. Here i counts through the variables that where omitted during reduction and n beeing the intrinsic variables. Again p counts through the intrinsic variables. The state space of lower dimension is called the intrinsic solution manifold V with $q(n, \mu_i) \in V$ [2].



(a) Evolving the BGK-model in the Sod shock tube in time is generated through evaluating the FOM in space x, velocity v and time t, which yields the solution $f(x, v, t)$. The operator Q is here the FOM described in chapter 2.



(b) Evolving the ROM of the BGK-model in the Sod shock tube in time through evaluating the ROM at n and μ_i, yields an approximation to the FOM solution $\tilde{f}$. The operator Q is here either POD or a autoencoder described in chapter 3.

Figure 4.1: Outline of the correlation between the FOM solution and the approximation obtained from the ROM.

MOR is partitioned into two successive phases called the *offline* - and the *online* phase. During the offline phase *snapshots* of a dynamical-system are generated through experiments or simulations of the full order model (FOM). The snapshots $F = \{f(t_1), ..., f(t_n)\}$ are created once, each representing one moment in time of the dynamical system. Thus in our case one needs a snapshot database of solutions $f(x, v, t)$ of the BKG-model in Sod's shock tube. Next the mapping Q is constructed such that $\tilde{f} = Q(f)$, for which $f(t_i) \approx \tilde{f}(t_i)$, reducing the dimensionality of the FOM solution as outlined before. During the online phase the reduced order model is evaluated and the error is estimated by

$$\mathrm{L}_2 = \frac{||f - \tilde{f}||_2}{||f||_2} \tag{4.1}$$

which is called the relative L_2-Error norm. The abbreviation L_2-Error is used from here on. Therefore the online phase may be described as a stage of independence from the full order model. Following [25] and [2], the success when building a ROM through linear reduction methods like the POD section 3.1, depends on a rapidly decaying Kolmogorov N-width. In particular advection dominated problems exhibit a slow decay of the Kolmogorov N-width as described in [25]. Thus yielding a need for non-linear methods like autoencoders described in section 3.2. The Kolmogorov N-width is given by

$$d_V(W) := \sup_{f \in W} \inf_{\tilde{f} \in V} ||f - \tilde{f}|| \tag{4.2}$$

and gives the worst best approximation error for elements of W. The convergence behaviour of the Kolmogorov N-width for advection dominated problems, especially when jump conditions are involved as in Sod's shock tube chapter 2, decays with

$$d_p(W) \leq \frac{1}{2} p^{-1/2}, \tag{4.3}$$

where p denotes the number of RB or intrinsic variables. Further insight in provided in [25].
Note that hereafter we will solely use the term intrinsic variables. The relevance of using non-linear reduction methods for MOR is often formulated in terms of a slow decaying Kolmogorov N-width. And even tough the BGK-model in Sod's shock tube describes an advection problem with jump discontinuities implies that linear reduction methods should fail, we will see in the following that this is only partly true in this case.

4.1 Offline phase and number of intrinsic variables

The FOM is the 1D BGK-model in Sod's shock tube for two levels of rarefaction, gratefully provided by Julian Köllermeier and the Departement of Mathematics at the RWTH Aachen. Sod's shock tube is discretized in space x with 100 nodes, in velocity v with 40 nodes for 25 time steps in time t, as presented in table 4.1 and fig. 2.3. One level can be viewed as a slip flow [13] with $Kn = 0.01$, hereafter referred to as **R**. The dilution up to this level of rarefaction is little though leading to inaccuracies when employing the common Navier Stokes equations. Therefore the NFS-equations (Navier-Stokes-Fourrier) could be used [26]. The other is situated in the continuum flow regime with $Kn = 0.00001$ for which the Navier-Stokes equations can be utilized without hesitation. An in-depth description of the BGK-model and Sod's shock can be found in chapter 2.
Hereafter the continuum flow will be referred to as **H**. Note that both **H** and **R** are three dimensional tensors comprising solutions $f(x, v, t)$.

Table 4.1: Problem setup for the BGK model in Sod's shock tube. The diaphragm is positioned at $x_d = 0.5025$. For the initial condition with $t = 0$ the gas is present at $x < x_d$ and absent for $x \geq x_d$.

Variable	Number of nodes i	Domain extension	Step size (uniform)
x	200	[0.0025,0.9975]	0.00499
v	40	[-10,10]	≈ 0.51282051
t	25	[0,0.12]	0.005

The reduction algorithms, introduced in chapter 3, require a distinct reshaping of the input data before they can be used. The preprocessed matrix for the FCNN as one batch $\mathrm{P_{FCNN}}$, is shown in eq. (4.4). Each row of $\mathrm{P_{FCNN}}$ represents $\mathrm{P_{FCNN},}_l = f(v, t_i, x_j)$, an example, that can be fed to the FCNN. Hence $x_j t_i = 5000$ samples can be acquired. POD can use $\mathrm{P_{FCNN}}$ as well as P_{FCNN}^T it's transposition as input.
The preprocessed matrix for the CNN, $\mathrm{P_{CNN}}$, is shown in eq. (4.5) with $\mathrm{P_{CNN,}}_l = f(x, t, v_k)$. Hence $v_k = 40$ examples can be obtained to be fed into the CNN. Because of the little amount of available examples per rarefaction level, it is decided to combine $\mathbf{H}$ and $\mathbf{R}$ to one dataset, yielding 80 available examples for training the CNN. This measure leads to one model, that has the abillity to generalize about the BGK-model for a variety of rarefaction levels in Sod's shock tube. Details can be found in appendix B. In the following, a distinction between $\mathrm{P_{CNN}}$ and $\mathrm{P_{FCNN}}$ is omitted, when referring to the preprocessed matrices

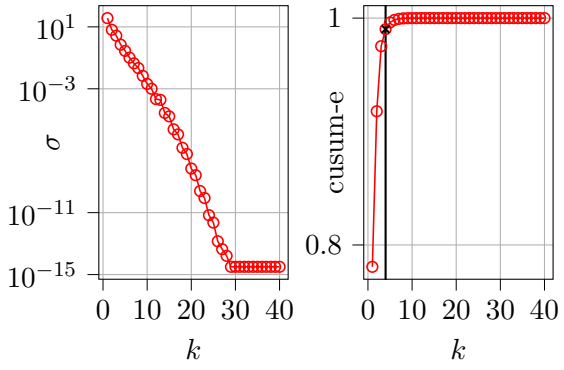
$$\mathrm{P_{FCNN}} = \begin{bmatrix} f(v_1,t_1,x_1) & \cdots & f(v_n,t_1,x_1) \\ f(v_1,t_1,x_2) & \cdots & f(v_n,t_1,x_2) \\ \vdots & \vdots & \vdots \\ f(v_1,t_1,x_n) & \cdots & f(v_n,t_1,x_n) \\ f(v_1,t_2,x_1) & \cdots & f(v_n,t_2,x_1) \\ \vdots & \vdots & \vdots \\ f(v_1,t_n,x_n) & \cdots & f(v_n,t_n,x_n) \end{bmatrix}, \qquad \mathrm{P_{CNN}} = \begin{bmatrix} n_{Filters}, & f(v_1,\mathbf{t},\mathbf{x}) \\ n_{Filters}, & f(v_2,\mathbf{t},\mathbf{x}) \\ \vdots & \vdots \\ n_{Filters}, & f(v_n,\mathbf{t},\mathbf{x}) \end{bmatrix}. \tag{4.4, 4.5}$$

With the FOM solution at hand it is possible to construct a mapping Q such that $Q(f) \approx \tilde{f}$. Again,for Q, POD and two autoencoders, the FCNN and the CNN, are employed. With considering appendix A and appendix B, the selection of hyperparameters and training for the FCNN and the CNN is discussed, providing fully trained and tuned FCNNs and a CNN, which are considered as given from now on.
Note that since this thesis focuses on the use of neural networks, POD acts as a benchmark. Consequently, in oder to contrast the number p of intrinsic variables of the autoencoders (sizes of the code layer) we will utilize POD as a reference framework. The intrinsic variables obtained from POD, the FCNN and the CNN, will be referred as $\mathbf{h}$ and $\mathbf{r}$, where the former describes the intrinsic variables when reducing $\mathbf{H}$ and the latter when reducing $\mathbf{R}$.
A first step is to perform a POD with $\mathbf{H}$ and $\mathbf{R}$ distinctively. The obtained singular values σ, as well as the cumulative energy (cusum-e), which is defined as

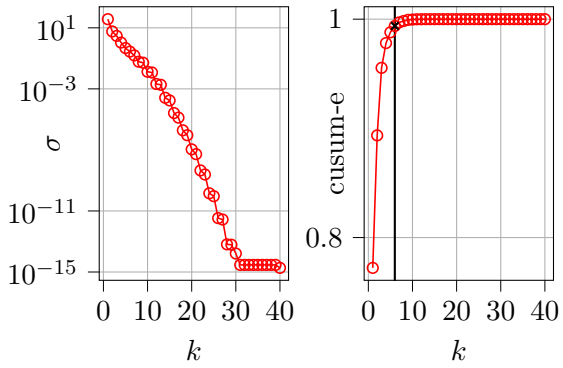
$$\text{cusum-e} = \frac{\text{cusum}}{\sum_i \sigma_i} \tag{4.6}$$

$$\text{with} \qquad (\text{cusum})_i = (\text{cusum})_{i-1} + \sigma_i \tag{4.7},$$

over the singular values, are shown in fig. 4.2. With a total of $p = 4$ intrinsic variables, a cumulative energy of over 99% can be achieved for **H**. The fourth singular value measures to a value of $\sigma_4 = 0.706$. The cumulative energy of the singular values of **R** arrives above 99% with $p = 6$ singular values. The sixth value is at $\sigma_6 = 0.275$. Thus a slight difference can be observed for both datasets when emplying POD.
We can link the decay of the Kolmogorov N-width in eq. (4.2) with the decay of the singular values, which invalidates the application of eq. (4.3) for the FOM solutions. The rate at which the singular values drop is approximately exponential in **H** and in **R** and in turn indicates a rapid decay of the Kolmogorov N-width. Hence supports the assumption that advection and sharp shock fronts do not appear predominantly. The eventough small, but dissimilar, decay rate of the singular values is a manifestation of the different rarefaction levels: With a decreasing number of particles present in Sod's shock tube, the lesser the probability of a single bulk macroscopic behavior emerges as seen in the full survey of the BGK model in chapter 2, leading to an expected increased number of intrinsic variables necessary to achieve similar L_2-errors.



(a) Sigular values σ over k number of singular values (left) and cumulative energy, here labeled as „cusum-e“ over k (right) for **H**. A black cross marker corresponds to over 99% cumulative energy.



(b) Sigular values σ over k number of singular values (left) and cumulative energy, here labeled as „cusum-e“ over k (right) for **R**. A black cross marker corresponds to over 99% cumulative energy.

Figure 4.2: Comparison of singular variables σ and cumulative energy for **H** and **R**. The decay of the singular values can be used to estimate the decay of the Kolmogorov n-width.

From a fluid mechanical point of view, the number of intrinsic variables for **H** in theory suffices with $p = 3$, as a slip-flow can be described in terms of three macroscopic quantities i.e. density ρ, macroscopic velocity u and total energy E_{tot}as described in eq. (2.11) to eq. (2.13) and in [1][6]. As mentioned before, does theoretically **R** require p to be of a larger number as more than a single maxwellian describe the microscopic velocities. Therefore **h** with $p = 3$ is employed and the FCNN is arranged in that way as a a starting value for the hyperparameter search. A starting value of $p = 5$ on the other hand is chosen for **r**. Note that this implies, that the CNN, which is trained with both rarefaction levels simultaneously, initially uses $p = 5$. The same value for p is chosen for the FCNN and **R**.
In oder to shed light into the performance of the autoencoders with a different number sizes for **h** and **r**, the size of the bottleneck layer of the autoencoders, which is the same as p, are varied in a next step.

To this end p is varied for POD, FCNN and CNN over $p \in \{1, 2, 4, 8, 16, 32\}$ for both rarefaction levels. Note that the neural networks needed to be trained for these experiments and that by changing p i.e. widening the bottleneck layer, a gain or loss of capacity occurs which can be connected to stability during training, see chapter 3 and [7]. Shown in fig. 4.3 is the outcome of said

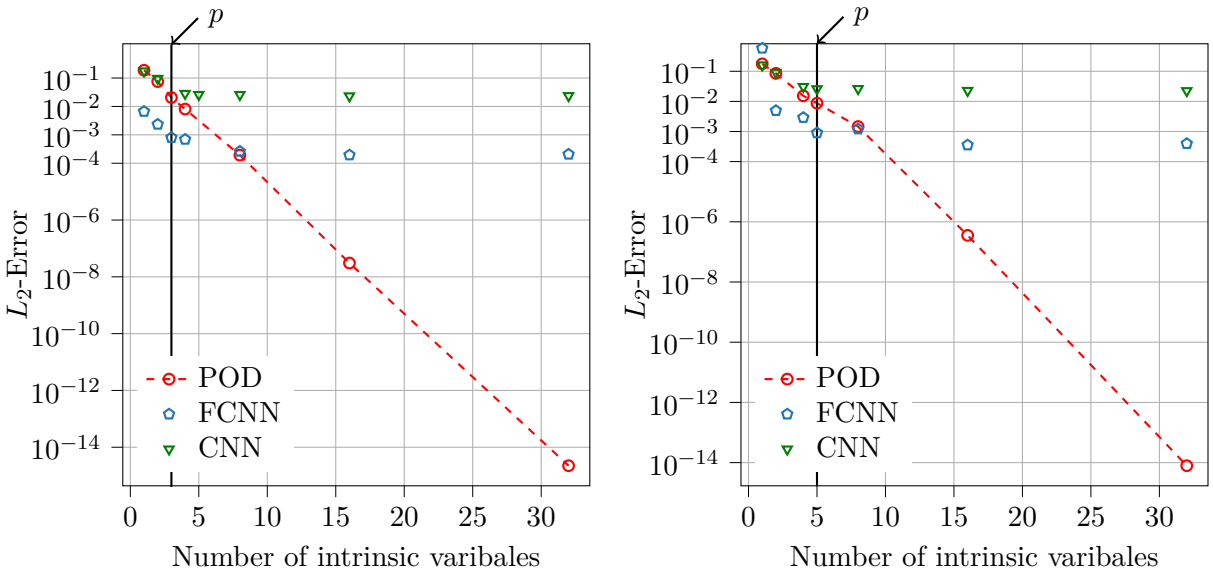


Figure 4.3: Variation of the number of intrinsic variables p over the L_2-Error for POD, FCNN and CNN. Results for **H** are displayed on the left and for **R** on the right.

experiments. The design for fig. 4.3 is taken from [2]. The loss of information when applying POD goes exponentially to zero with increasing p which is not suprising when consulting the *Eckard-Young Theorem* provided in eq. (3.4) taken from [5]. The left plot of fig. 4.3 displays the results for **H** with $p = 3$ the estimated size of **h**, emphasized with a black line. The L_2-error of the FCNN first drops until $p = 3$ reaching $L_2 = 0.0008$ to then drop further until $p = 8$ with $L_2 = 0.00026$. Afterwards the L_2 stagnates with a best value of $L_2 = 0.00019$ at $p = 16$. Interestingly, the biggest improvement can be observed until $p = 3$.
The L_2-error of the CNN, that was trained with both rarefaction levels, drops until $p = 4$ reaching $L_2 = 0.028$. Afterwards the L_2-error stagnates reaching it's best value of $L_2 = 0.023$ at $p = 16$ and $p = 32$. One can assume, that the value of the L_2-error is approximately the same for $p = 3$ and $p = 4$, as it is for the FCNN. Therefore the CNN, as well as the FCNN, seem to reflect the assumptions for p considering **H**. While the CNN and POD are congruent for $p = 1$ and $p = 2$ with POD outperfoming the CNN afterwards, the FCNN is only outperformed by POD after $p = 8$, where the L_2-error of both algorithms meet.
Moving forward, to assess the value of p for **R** consider the right plot of fig. 4.3. Again $p = 5$ is highlited with a black line to indicate it's assumed value. The L_2-Error of the FCNN begins as the highest of all three algorithms with $L_2 = 0.58$ at $p = 1$. Afterwards the L_2-error plummets outperforming POD and the CNN until $p = 8$. After $p = 8$, where POD and the FCNN meet with $L_2 = 0.012$ and $L_2 = 0.014$ respectively, POD outperforms both algorithms. From $p = 4$ to $p = 5$ the L_2-error of the FCNN drops from $L_2 = 0.0029$ to $L_2 = 0.0009$. The increase in L_2-error moving from $p = 5$ to $p = 8$ is not an result of overfitting as seen in appendix B, and therefore can only be explained with a bad initialization point of the networks free parameters θ. A continued widening

of the bottleneck layer results in the lowest error of $L_2 = 0.00035$ at $p = 16$ for the FCNN.
Resulting L_2-error values of POD and the CNN match for $p = 1$ and $p = 2$. Thereafter the error drops to $L_2 = 0.03$ and $L_2 = 0.026$ for $p = 4$ and $p = 5$ respectively. Continuing to widen the bottleneck layer one can observe, that the L_2-error stagnates. Nevertheless, the lowest error is reached with $L_2 = 0.022$ at $p = 16$ and $p = 32$.
The variation of p shows that for the assumed values of $p = 3$ and $p = 5$ the FCNN outperforms POD. Additionally, the performance of both autoencoders increases up to those points, which accentuates the previously made assumptions for p. Nonetheless, those are not the lowest values the FCNN, and with some limitations also the CNN, can reach. Until $p = 8$ the FCNN reaches it's limitations for surpassing POD. Hence, depending on p either POD or the FCNN have the ability to surpass each other. In order to evaluate the interpretability of all three algorithms, $p = 3$ and $p = 5$ is chosen for sizes of $\mathbf{h}$ and $\mathbf{r}$ respectively.

The following chapter covers the discussion of results obtained from POD, the FCNN and the FCNN. Additionally the evaluation of new states of the FOM is offered, which can be viewed as the online phase of MOR.

5 Results

This chapter covers the evaluation of reconstructions $\tilde{f}$ obtained from the FCNN and the CNN through a comparison against reconstructions obtained from POD. Additionally, an analysis of the interpretability of the intrinsic variables **h** and **r** is provided. This chapter concludes with the attempt to create new states of the FOM with the FCNN and the CNN, which can be viewed as an online phase of MOR.
To begin with the benchmarking of POD and neural networks, the number of parameters to obtain $\tilde{f}$ is contrasted. Beforehand solely the number of trainable parameters that compose both neural networks were called θ. For this comparison, θ is extended to also include all elements of the left and right singular vectors as well as the singular values of POD. Additionally the amount of intrinsic variables used for reconstruction is set to $p = 3$ and $p = 5$ for **H** and **R** respectively. An exception is the CNN, that uses $p = 5$ independent from rarefaction level. A summary is provided in table 5.1. POD uses with 15129 and 25225 parameters to reconstruct **H** and **R** respectively the

Table 5.1: Amount of parameters θ used to reconstruct f, number of intrinsic variables p and the corresponding L_2-Error for POD, FCNN and CNN.

Algorithm	Parameters θ		Int. variables p		L_2-Error	
	H	**R**	**H**	**R**	**H**	**R**
POD	15129	25225	3	5	0.0205	0.0087
FCNN	2683	3725	3	5	0.0008	0.0009
CNN	8246	8246	5	5	0.025	0.027

largest amount of parameters of all three algorithms. These yield L_2-errors of 0.0205 and 0.0087 respectively. Interestingly, the elevation of p amounts to an increase of paramters by approximately 1.7 which is comparable to the FCNN with an approximate increase of 1.4. The FCNN, which holds the best L_2-errors of 0.0008 and 0.0009 for **H** and **R** respectively, does so with the least amount of parameters. For reconstructing **H** solely 2683 and for the reconstruction of **R** solely 3725 parameters are used, which is a fraction of the need for POD. The second most populous alhorithm is the CNN which uses 8246 parameters for both rarefaction levels. The resulting L_2-errors with 0.025 for **H** and 0.027 for **R** are the largest of all three algorithms.

Next a qualitative analysis with actual reconstructions is presented. For this purpose the L_2-error over time t, seen in fig. 5.1, is used to localize the most challenging snapshot for each algorithm.
With POD and the CNN the last timestep at $t = 0.12s$ for both rarefaction levels is the most rich in the $L2$-Error. In contrast, the FCNN does not show a distinct time dependence of the L_2-error. Nonetheless, struggles at the onset at around $t = 0.005s$ with **H** and around $t = 0.005s$

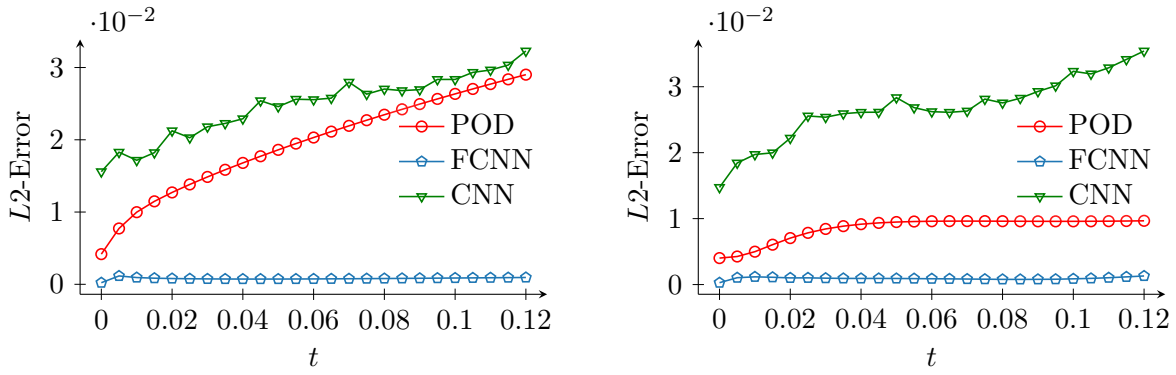


Figure 5.1: L_2-error over time for POD, FCNN and CNN. Results for **H** are displayed on the left, the results for **R** are displayed on the right.

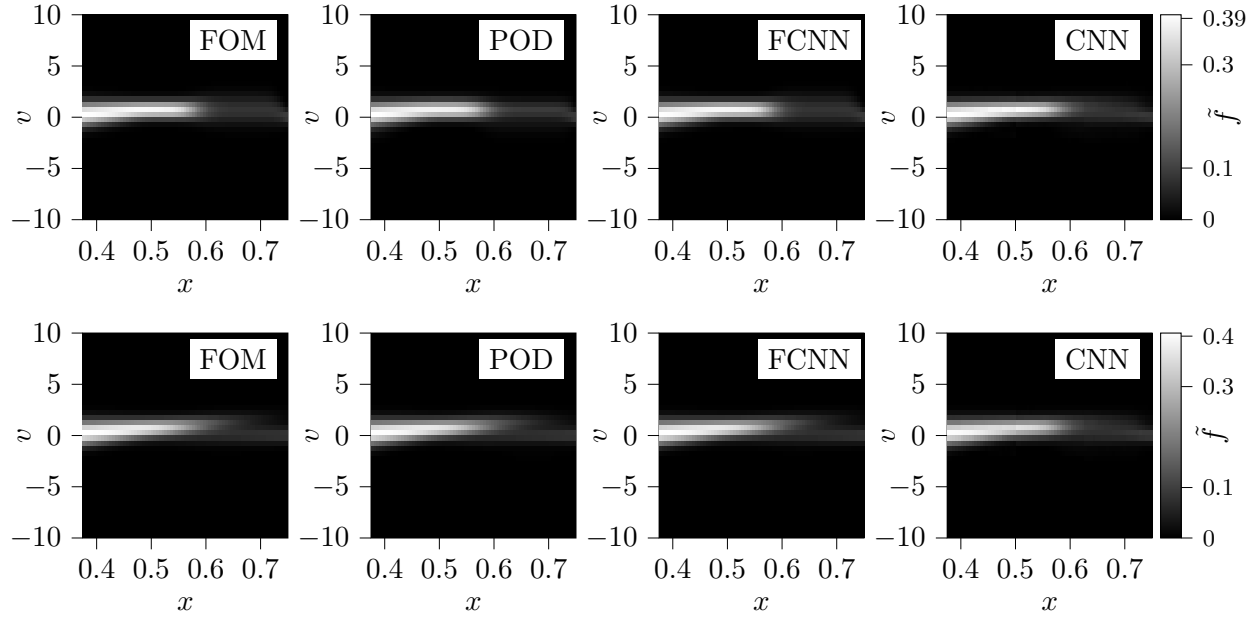


Figure 5.2: Comparison of the FOM solution f with three reconstructions $\tilde{f}$ obtained from POD, the FCNN and the CNN. Reconstrucions are shown at $t = 0.12s$ for $x \in [0.375, 0.75]$. Case **H** is displayed in the top row, **R** in the bottom row. The colobars reference f and $\tilde{f}$.

and $t = 0.0115$ (in the beginning and at the end) with **R** can be observed for the FCNN. Examples of reconstructions $\tilde{f}(x, v, t_i)$ with $t_i = 0.12s$ and $x \in [0.375, 0.75]$ are given in fig. 5.2.

The FOM solution viewed as $f(x, v, t_i)$ has been introduced in chapter 2. There, $f(x_j, v, t_i)$ is the probability distribution of the microscopic velocities v at point x_j in space at one moment t_i in time for a gas. With this in mind, a qualitative comparison between the three algorithms can be made considering the rendition of the velocity probabilities. Starting with **H**, seen in the top row of fig. 5.2 one can observe that $\tilde{f}(x, v, t_i)$ starting around $x = 0.6$ gets defective for POD and the CNN. Noteworthy, here the probability distribution is thinner as the original with POD. This in turn leads to errors in the temperature T once passing $x \approx 0.6$. Prominent qualitative deviations

using the CNN are especially blurriness/pixelation of $\tilde{f}(x, v, t_i)$ after $x \approx 0.6$. In contrast the FCNN seems to reproduce the FOM solution almost exactly.
Continuing with a row further down in fig. 5.2 and therefore with **R**. The FCNN seems to reproduce the FOM solution without any visible drawback. Also POD seems to reproduce all important structures, except after $x \approx 0.7$ around the contact discontinuity, some values for velocities with $v > 0$ appear to be missing. Again the CNN struggles with blurriness making $\tilde{f}$ for both rarefaction levels look largely similar.

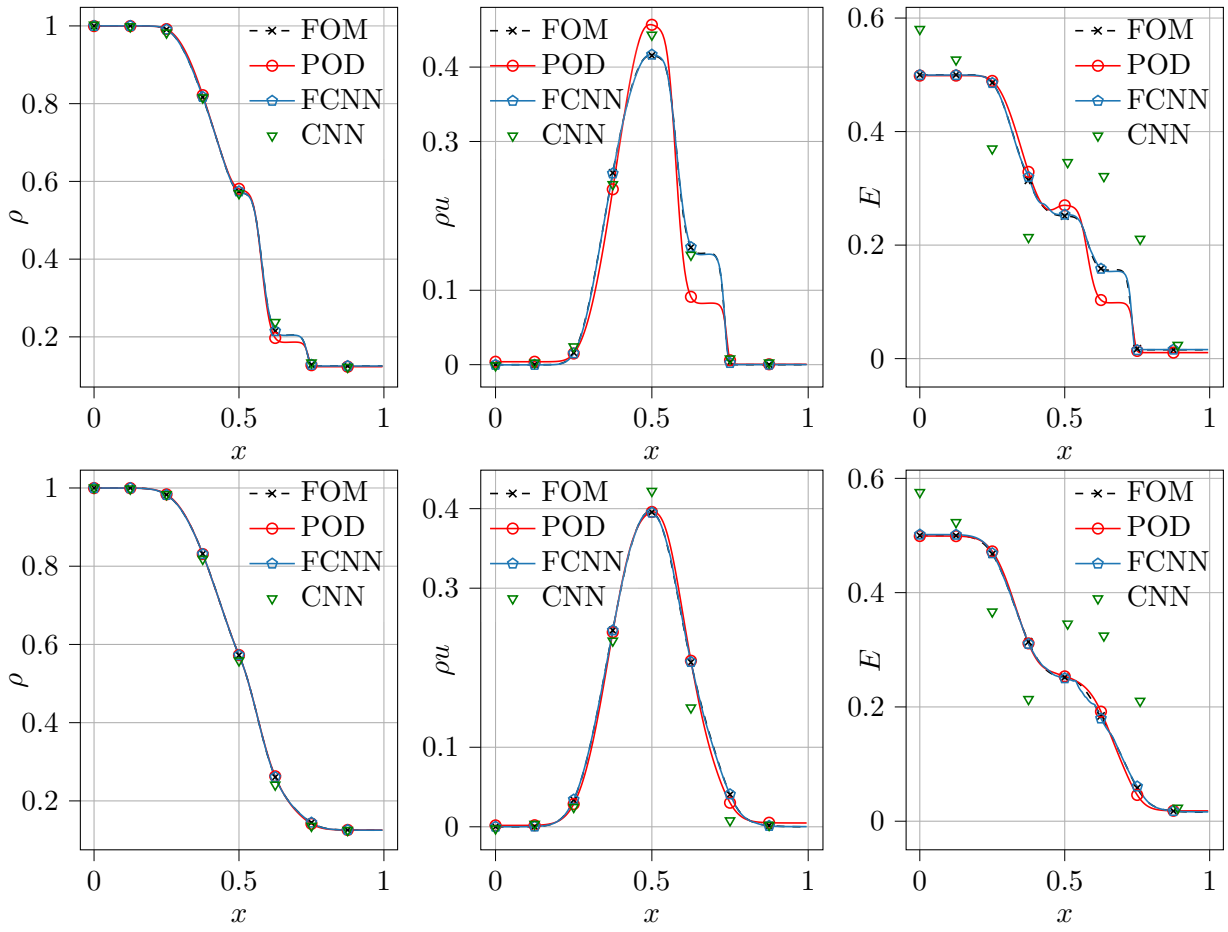


Figure 5.3: Matching of macroscopic quantities ρ, ρu and E reproduced by POD, the FCNN the and CNN with macroscopic quantities computed from the FOM. Top row shows results for **H**, bottom row for **R** at time $t_i = 0.12s$. CNN is displayed with marks only because of trembles in the signal.

Loss of information described above can unfold in severe mistakes in ρ, ρu and E, the macroscopic quantities, as displayed in fig. 5.3. Examining the macroscopic quantities enables a detailed look on the reconstruction errors. Features of the macroscopic quantities are expressed in terms of rarefaction wave, contact discontinuity and height as well as position of the shockfront. For a detailed elaboration of these terms see chapter 2. Following the structure in the preceding figures, macroscopic quantities of **H** are displayed in the top row and for **R** in the bottom row of fig. 5.3. First the reproduction of the macroscopic quantities ρ and ρu obtained by the FCNN match the FOM solution exact for both levels of rarefaction **H** and **R**. Interestingly, despite the overall

impressive performance of the FCNN regarding the small number of parameters it uses, the total energy shows small deviations around the tail of the rarefaction wave for **H** and somewhat severe errors at the transition form rarefaction wave to shock front for **R**. Second the CNN produces trembles in ρu and especially in E which is why it's shown with marks only. The macroscopic quantities reproduced by the CNN show unmissable, it's inabillity to differentiate between **H** and **R**. Specifically, seem the macroscopic quantities for **R** appear to be a copy of the the ones for **H**. Additionally, considering **H**, one can observe, that the momentum ρu holds small errors for the tail of the rarefaction wave as well as the contact discontinuity. The value for the tip of the shockwave exceeds, comparable with POD, the exact solution. Third POD performs better on **R**, which is not suprising considering the the difference in number of used parameters, holding only small deviations of the contact discontinuity and the shockwave for the momentum ρu and the total energy E. The density ρ matches the FOM solution exact. Pronounced deviations from the FOM solution occur using POD on **H**. The density ρ undercuts the original shockwave. The momentum ρu heavily exceeds the tail of the rarefaction wave and in the same extent undercuts the contact discontinuity. Hence in the total energy E the same is observable for the tail of the rarefaction wave and the contact discontinuity.

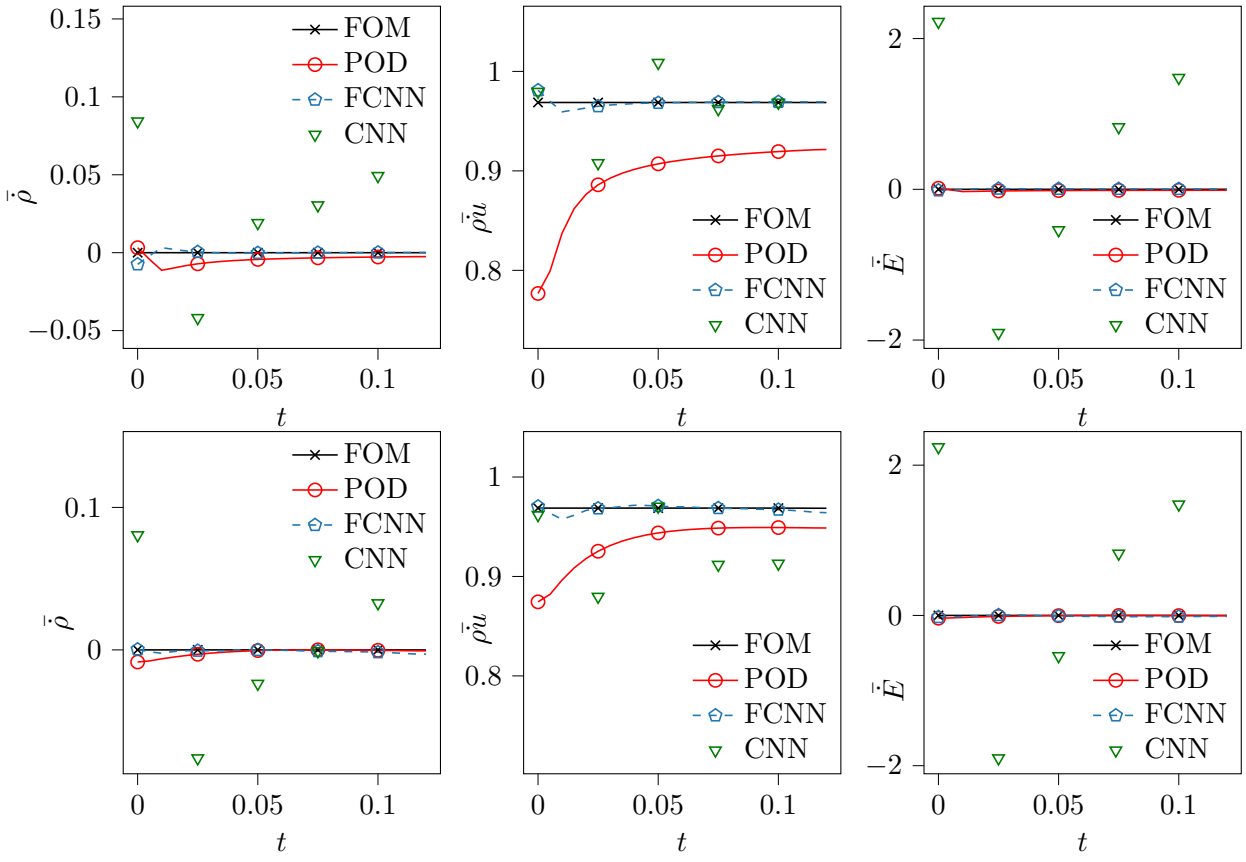


Figure 5.4: Comparison of the conservative properties of reconstructions obatined from POD, the FCNN and the CNN against the conservative properties of the FOM solution using the temporal mean.

The physical consistency of $\tilde{f}$, in terms of conservation of mass momentum and energy is a critical

criteria for it's validity. Hence conservation properties are analyzed in the following. In that respect, the temporal mean over the time derivative, which can be calculated exemplary for ρ with

$$\frac{\mathrm{d}}{\mathrm{d}t}\int \rho(x,t)\,\mathrm{d}x\Delta t = \bar{\dot{\rho}}, \tag{5.1}$$

of the macroscopic quantities is employed. Figure 5.4 shows the conservation of mass, momentum and total energy as a temporal mean for **H** in the top row and for **R** in the bottom row.
Conservation of mass is met using the FCNN, except for small deviations at the outset for both cases **H** and **R**. Similarly, does POD meet conservation of mass for **R**. The erroneous **H** case losses mass at the onset and gains mass towards the end with POD. Conservation of momentum meets the FOM solution, except for minor gains and losses, after $t \approx 0.03s$ using the FCNN for both cases **H** and **R**. POD gains momentum of 0.13 for **H** and 0.07 for **R**. The conservation of total energy is met for **H** and **R** using POD and the FCNN. Finally the reconstructions of the CNN do not conserve mass, momentum nor total energy. All conservative properties behave comparable to a sawtooth wave. A gain and loss of either of the quantities can be observed.

Because of the black-box nature of neural networks the question of interpretability often arises when working with them [27]. Especially benchmarking neural networks for model order reduction with POD asks for the evaluation of the interpretability of the intrinsic variables. Owing to the tremendous quality of POD, which lies in "the physically interpretable decomposition"[5][p.375] of the input data, as stated in chapter 3.
Following the deduction, that **H** can be completely described in terms of three macroscopic quantities and that **R** is describable in a similar way, it is put to test if the intrinsic variables **h** and **r** show any similarities, if not match any macroscopic quantity. To this end two supplementary macroscopic quantities namely the temperature T and macroscopic velocity u are added to the three macroscopic variables. In fig. 5.5 and fig. 5.6 those are depicted first over the whole domain of x and t and for two specific timesteps $t = 0.055s$ and $t = 0.12s$ for **H** and **R** respectively. Similarily are the intrinsic variables **h** and **r** with

$$[h_0(x,t),\ldots,h_p(x,t)] = \mathbf{h} \tag{5.2}$$ and $$[r_0(x,t),\ldots,r_p(x,t)] = \mathbf{r} \tag{5.3},$$

depicted in fig. 5.7 and fig. 5.8 respectively.
Strikingly, most intrinsic variables appear to be a sort of linear combination of the five intrinsic variables. In particular **h**. For example do the rarefaction wave, shock wave and contact discontinutity, that can be identified in h_0 reflect a combination of those found in the density ρ and the total energy E. Furthermore does h_1 seem to be a mirror image of the momentum ρu, where beginning end values are inspired by the temperature T. Then again T plays a more pronounced role in h_2, where it's fluctuation appears. The peak of h_2 could be guessed from the peak of the macroscopic velocity u with it's bump at $t = 0.12$.
A clear identification of macroscopic quantities that can be seen in **r** is possible for r_3, which reflects the shape of the density ρ. Moreover, the peak of the velocity u can be rediscovered in the through of r_0. For other intrinsic variables of **r** namely r_1, r_2 and r_3 a clear discernability of macroscopic quantities is not observed. It seems rather that those constitute abstract information about the rarefied flow. Nonetheless it can't be verified, that also those can be created through linear combinations of the intrinsic variables.

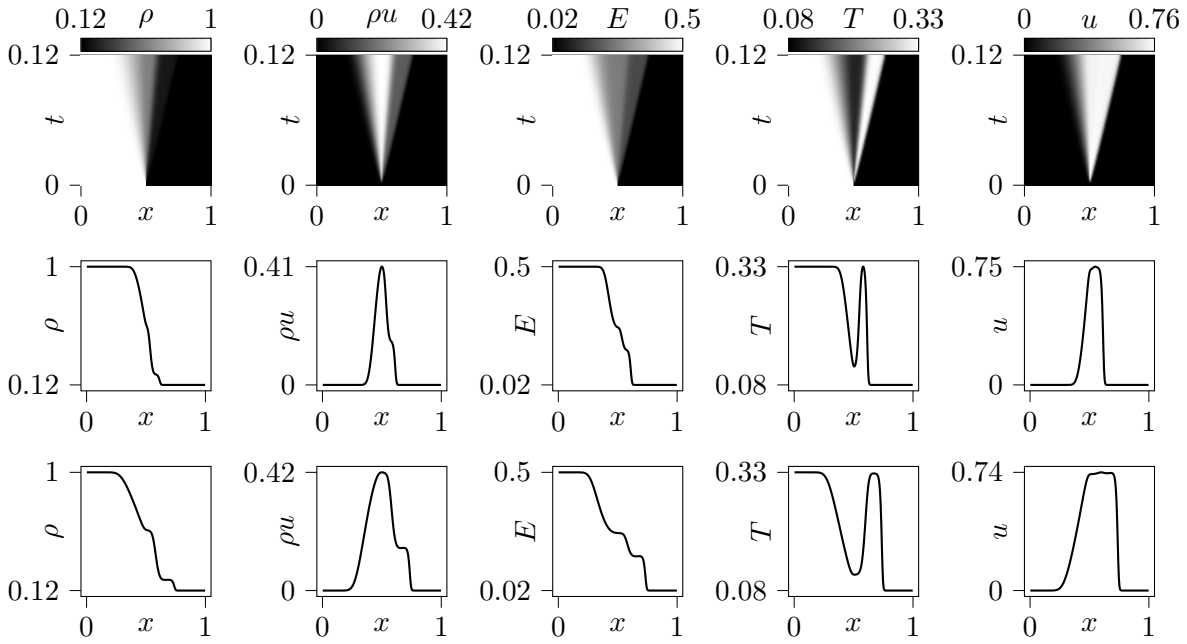


Figure 5.5: Macroscopic quantities of **H**. Density ρ, momentum ρu, total energy E, teperature T and velocity u over time t and space x in the top row and specifically at $t_i = 0.055s$ and $t_i = 0.12$ in the middle and bottom row respectively.

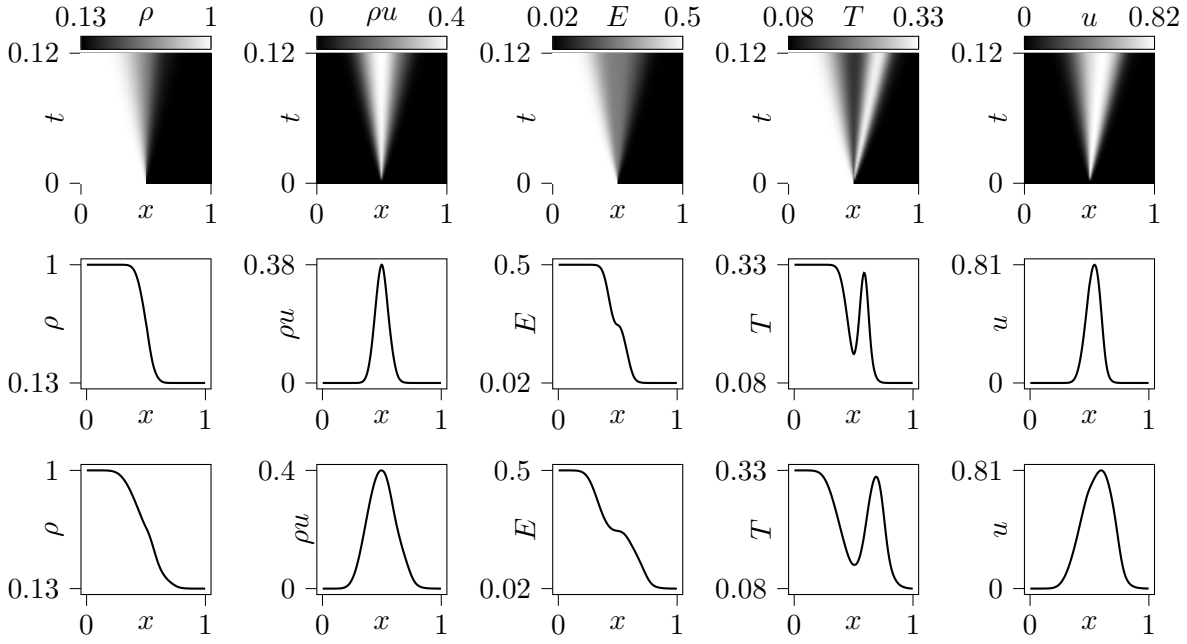


Figure 5.6: Macroscopic quantities of **R**. Density ρ, momentum ρu, total energy E, teperature T and velocity u over time t and space x in the top row and specifically at $t_i = 0.055s$ and $t_i = 0.12$ in the middle and bottom row respectively.

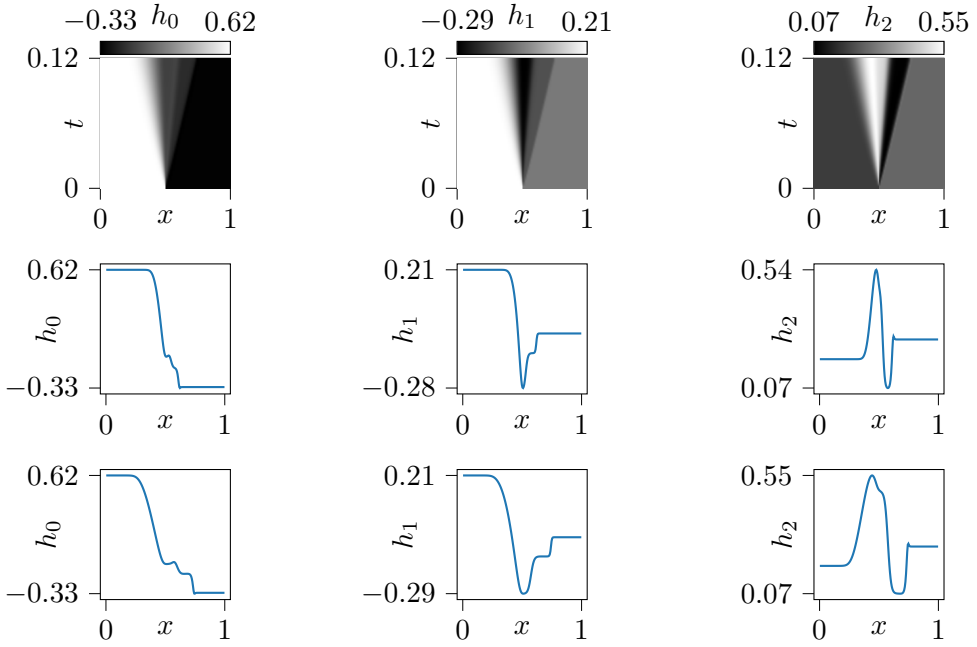


Figure 5.7: Intrinsic variables $h_0(x,t)$, $h_1(x,t)$ and $h_2(x,t)$ of $\mathbf{h}$ obtained from the FCNN. Top row depicts $\mathbf{h}$ over the whole (x,t) domain, middle and bottom row for $t = 0.055$ and $t = 0.12$ respectively.

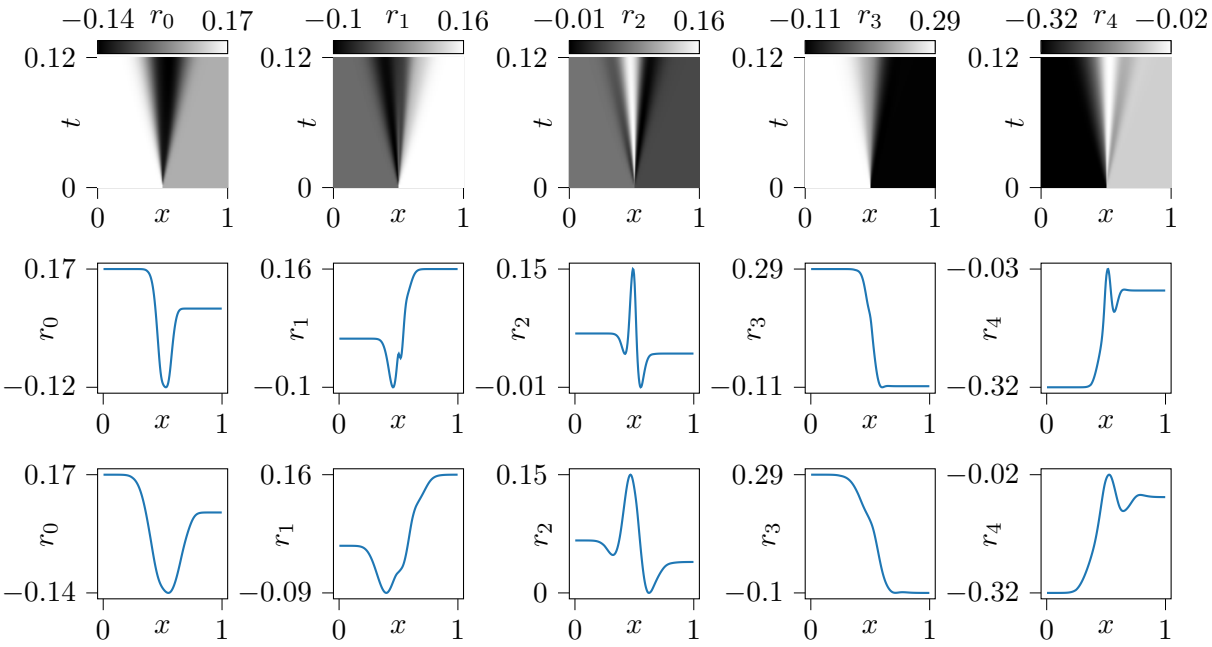


Figure 5.8: Intrinsic variables $r_0(x,t)$, $r_1(x,t)$, $r_2(x,t)$, $r_3(x,t)$ and $r_4(x,t)$ of $\mathbf{r}$ obtained from the FCNN. Top row depicts $\mathbf{r}$ over the whole (x,t) domain, middle and bottom row for $t = 0.055$ and $t = 0.12$ respectively.

The intrinsic variables extracted from the CNN and POD are shown in fig. 5.9. They are only dependent on v with

$$[h_1(v), \dots, h_p(v)] = \mathbf{h} \qquad (5.4) \qquad \text{and} \quad [r_n(v), \dots, r_p(v)] = \mathbf{r} \qquad (5.5)\cdot$$

Third and fourth row of fig. 5.9 show the first three and five POD modes of **H** and **R** respectively. As expected, the first three modes of both rarefaction levels do not differ significantly to the same extend as the bulk dynamic answer of the two gas flows to the test case doesn't. Pronounced differences of **h** and **r** extracted from the CNN, senn in the first and second row of fig. 5.9, are also not observed. This aligns with previous findings that the CNN fails to differentiate between **H** and **R**. Qualitatively do the intrinsic variables of the CNN show less complexity compared to those of POD, nonetheless a rough analogy between the two can be observed. Especially h_0/r_0, the first intrinsic variables, of the CNN resemble those of POD. Furthermore can a small through in h_1/r_1 of the CNN be identified which is present, although much more pronounced, in h_1/r_1 of POD. Worth mentioning is that h_0 and r_0, h_1 and r_1 as well as h_4 and r_4 of the CNN encompass the largest peaking of all intrinsic variables.

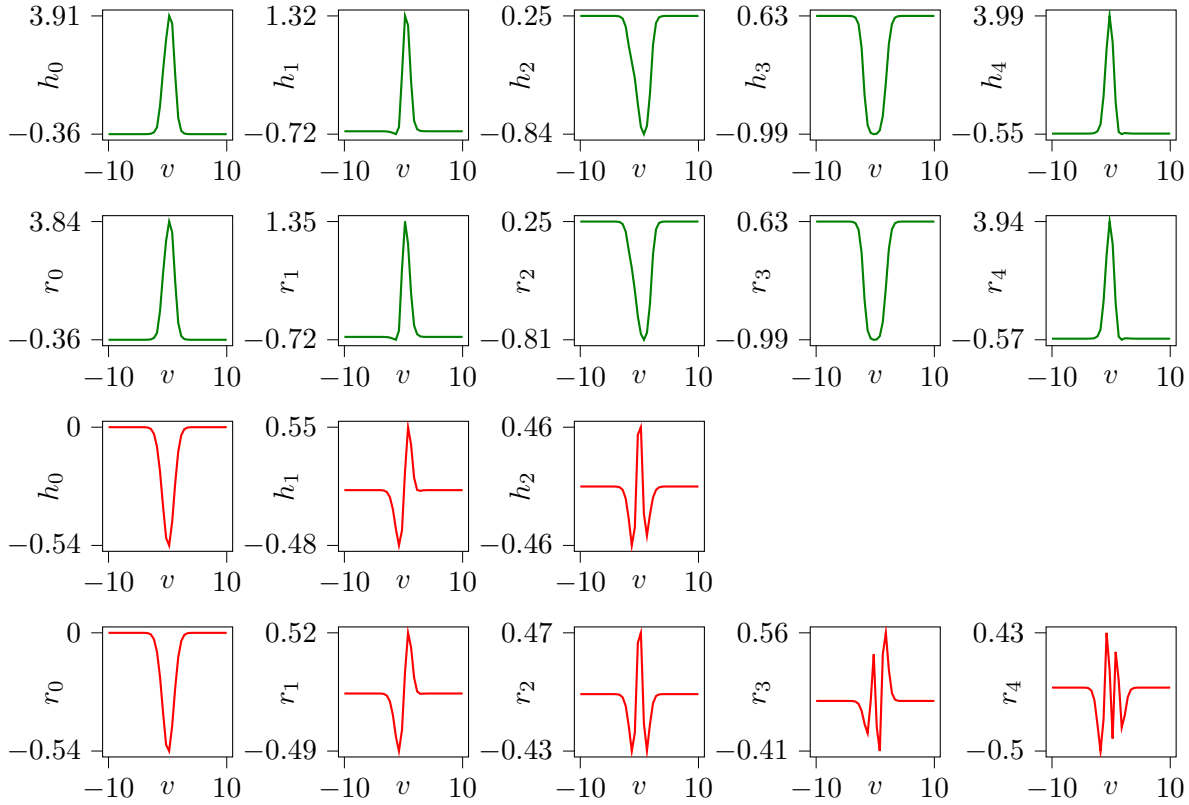


Figure 5.9: Intrinsic variables $h_0(v)$, $h_1(v)$, $h_2(v)$, $h_3(v)$ and $h_4(v)$ of **h** in the top row and $r_0(v)$, $r_1(v)$, $r_2(v)$, $r_3(v)$ and $r_4(v)$ of **r** of in the second row extracted from the CNN. Third and fourth row show $h_0(v)$, $h_1(v)$, $h_2(v)$ of **h** and $r_0(v)$, $r_1(v)$, $r_2(v)$, $r_3(v)$ and $r_4(v)$ of **r** extracted from POD respectively.

With POD one usually exploits the intrinsic variables within a Galerkin framework as in [6] to produce new states. The same can be done with the intrinsic variables obtained from autoencoders

as in [2]. Both won't be discussed in this contribution. Rather new states are obtained by performing an interpolation in time t of $\mathbf{h}$ and $\mathbf{r}$. This approach tests a different kind of ability to generalize about the FOM solution. So far there have been ested two kinds of generalization: The first can be encountered during training, where a split into train- and validation set probes the ability to fit unseen examples. A second approach into generalization is tried with the CNN, where both rarefaction levels as training examples probes the ability to fit both in the same model. The third insight into the ability to generalize is tested using the FCNN. By the temporal interpolation in the intrinsic variables it is probed if new states can be generated that meet the condition of MOR that the distance $||f - \tilde{f}||$ is small.
For this thesis an additional solution $\mathbf{H}^*$ of the BGK model in Sod's shock tube with $Kn = 0.00001$ is provided with a temporal resolution of 241 snapshots. This resolution is 9.64 times finer than the original one. Hence an interpolation using cubic splines in the temporal axis of $\mathbf{h}$ is performed. Thus resulting intrinsic variables called $\mathbf{h}^*$, which, needless to say, also have a temporal resolution of 241, are then fed into the decoder of the FCNN generating $\widetilde{\mathbf{H}}^*$. A comparison of $\mathbf{H}^*$ against $\widetilde{\mathbf{H}}^*$ as truth against prediction in terms of macroscopic quantities density ρ, momentum ρu and total energy E is provided in fig. 5.10.

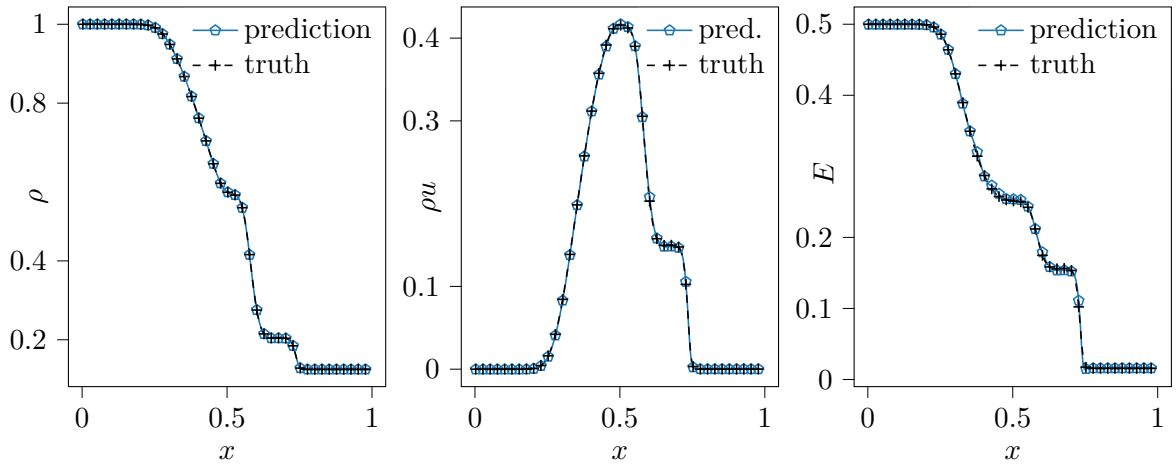


Figure 5.10: Macroscopic quantities density ρ, momentum ρu and total Energy E, displayed in this order, at time $t = 0.12$ against predictions generated from interpolation. Temporal interpolation with cubic splines is perofrmed in $\mathbf{h}$ from 25 snaptshots to 241 snapshots.

Figure 5.9 shows that the generated snapshots match the macroscopic quantities almost exactly, similar to the results obtained without interpolation. Nonetheless, the resulting L_2-error for $\widetilde{\mathbf{H}}^*$ increases to $L_2 = 0.0012$, which is 1.5 times higher than the original L_2-error.

In conclusion, the error over time for the CNN is disordered, showing gain and subsequent loss of information from one timestep to another. The CNN performs slighly better with $\mathbf{H}$ than with $\mathbf{R}$. What is hidden when looking at reconstructions becomes visible when verifying over the macroscopic quantities. Reconstructions obtained from the CNN show oscillations in the momentum ρu and the total energy E. On top of that the CNN does not meet conservation in any of the conservative properties. All of this together makes the CNN with this setup, especially the access to only 40 samples, unsuited for building a ROM. Next POD shows a noticeable increase in loss of information over time for both cases $\mathbf{H}$ and $\mathbf{R}$. Reconstructions of the last timestep as well as

the macroscopic quantities at that time reveal that the POD is unsuited for building a ROM with the **H**case. However, with **R** POD shows only slight deviations from the FOM solution. Taking conservative properties of the reconstructions obtained from POD into consideration only underlines aforementioned findings. Ultimately POD could be taken for building a ROM with **R**. Finally the the FCNN is the best performing model out of the three for both cases **H** and **R**, while the performance for **H** is slightly better than that for **R**. The error over time reveals a constant low loss. Only at the first time steps a noticeable loss of information is observed. Reconstructions of the last time step and the macroscopic quantities at that time are close to exact to the FOM solution for both cases **H** and **R**. The conservation of the macroscopic quantites only emphasize the proximity to the FOM solution. In total the FCNN is suited for building a ROM with both cases **H** and **R** and will be taken further into the online phase.

5.0.1 Discussion and Outlook

One reason of the lacking ability of the CNN is the small number of samples, as described in appendix B and chapter 4. The resulting trembles of the signal when calculating the macroscopic quantities is due the kernel approach of the CNN. During reconstruction the resolution of the output image is bounded by the size of the kernel, which leads to pixelation.
What we see here is the known drawback of POD. Sharp fronts and especially advection dominated problems lead to a fast decaying kogolomorov n-width. These problems need a nonlienar ansatz, as described in chapter 4.

Appendix

A Hyperparameters for the Fully Connected Autoencoder

The finding of appropriate hyperparameters for the fully connected autoencoder is described here. The hyperparameters include number of layers i.e. depth, number of nodes per hidden layer i.e. width, batch size and non-linear activation functions, number of epochs for training and learning rate. The experiments are evaluated through the validation error which estimates the model's ability to generalize, the training error which estimates the optimization to training data and the L_2 as described in chapter 4. Both validation- and training error are described in chapter 3 and provide information about under- and overfitting. Moreover the validation error is the essential metric for validating a model's performance. The L_2 on the other hand gives an estimate of how well the model performs on the whole dataset hence is used as a comparative metric against POD. To start with a working model first a guess about the some initial hyperparameters is done, which are summarized in Table A.1. These include a mini-batch size of 16, the width of the bottleneck layer is 3 and 5 for **H** and **R** respectively and a learning rate of 0.0001. Activation LeakyReLU is applied for the outputs of the input- and any hidden layer which does not output the code, referred to as activations hidden. Tanh is applied for the output of the last hidden layer in the encoder which outputs the code, referred to as activation code. A visualization of the activation scheme is provided in fig. A.1. Moreover 2000 initial number of epochs are used. This might seem exaggerated but is justified by the little amount of input data and the small size of the network which yiels a fast training.

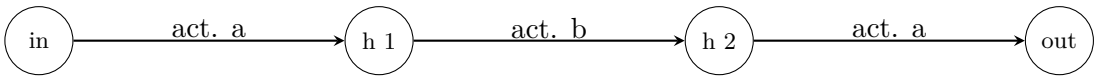


Figure A.1: Scheme of a network with four layers showing where the activations are placed. Activation act. a activates the output of the input layer as well as the output of any other hidden layer except the output of the last hidden layer in the encoder which is activated through act.b and outputs the code.

Table A.1: Initial selection for batch size, bottleneck size, number of epochs, learning rate and applied activation functions.

Mini-batch size	Intrinsic dimensions	Epochs	Learning rate	Activations hidden/code
16	3/5	2000	0.0001	LeakyReLU/Tanh

Five designs for finding an optimal number of layers i.e. depth are run. The hidden layers are designed to halve the input at each step. Not within this scope is the bottlenck layer which has a

fixed size of 5 and 3 for **R** and **H** respectively and the first and last hidden layer (after the input layer and before the output layer). Those should provide an abstraction without shrinking the incoming data. Note that this design feature was validated in an initial exploration cycle which is not included in this thesis. The five designs range from 10 layers in (a) to 2 layers in (e) always taking one layer away from encoder and decoder hence decreasing by two layers and are as follows:

(a) 10 layers with layer widths: 40, 40, 20, 10, 5, 3/5, 5, 10, 20, 40, 40.

(b) 8 layers with layer widths: 40, 40, 20 , 10, 3/5, 10, 20, 40, 40.

(c) 6 layers with layer widths: 40, 40, 20 , 3/5, 20, 40, 40.

(d) 4 layers with layer widths: 40, 40, 3/5, 40, 40.

(e) 2 layers with layer widths: 40, 3/5, 40.

The model's depth is determined in a primary step, as it sets a consequential part of the model's representational capacity and therefore can initiate over- and underfitting at an early stage in the hyperparameter search. The results of the experimentation are shown in fig. A.2 and table A.2 for both rarefaction levels.
For **H** the lowest validation error of 7.74×10^{-8} and an L_2 of 0.0031 is reached with 4 layers, hence

Table A.2: Results for the variation of depth. Given are minimum values of training and validation error as well as the L_2. The minima where reached around the last 50 epochs of the training. The L_2 is evaluated with the model at the last epoch.

Depth	Minimum training error		Minimum validation error		L_2	
	H	**R**	**H**	**R**	**H**	**R**
10	1.53×10^{-7}	5.96×10^{-7}	2.22×10^{-7}	5.19×10^{-7}	0.0048	0.0091
8	1.17×10^{-7}	2.05×10^{-7}	1.58×10^{-7}	2.32×10^{-7}	0.0041	0.0054
6	9.76×10^{-8}	1.40×10^{-7}	1.49×10^{-7}	1.72×10^{-7}	0.0038	0.0045
4	6.29×10^{-8}	1.52×10^{-7}	7.74×10^{-8}	1.61×10^{-7}	0.0031	0.0048
2	1.29×10^{-6}	3.29×10^{-6}	1.37×10^{-6}	3.42×10^{-6}	0.0136	0.0217

constitutes the best performing design out of the five. Additionally, as seen in fig. A.2(left), does a design exceeding 4 layers results in a slight overfitting after around 500 epochs. Less than 4 layers do not reach the validation error and L_2 of the other designs, yielding the conclusion, that the capacity is too low. Overfitting occurs with 4 layers only after the 1000th epoch and is less than with the other three models that show overfitting.
For **R** the lowest validation error of 1.61×10^{-7} is reached again with 4 layers. On the other hand the lowest L_2 of 0.0031 and the lowest training error of 1.40×10^{-7} are reached with 6 layers. Contrary to the afore discussed case the training error and L_2 are of lower magnitude for 6 layers, except for the validation error. Looking at fig. A.2(right), we observe that the model with 6 layers starts to overfit after the 1500 epochs, yielding a decreasing training error and a stagnating validation error. Hence the model improved in the optimization task which additionally improves the L_2. It's generalization ability, measured by the validation error, did not improve and is greater than the validation error reached with 4 layers. This concludes, a model with 4 layers constitutes the best performing design out of the five.

Qualitatively the overall training for both rarefaction levels is very stable. Training and validation error do not diverge exessively and converge early in training. Separation of training and validation error occurs prominently for the hydrodynamic solution. This is thought to be connected to the emersion of sharp shock fronts towards the end of the simulation. This increases variance in the whole dataset and therefore also in the training- and validation set.
The number of epochs is now doubled to 4000 epochs because the lowest validation error was achieved towards the end of the training in the previous experiments. Again this is justifiable as one epoch takes less than 1s to finish and no prominent overfitting is observed. The width of the

Table A.3: Results for the variation of width. Given is the minimum value of validation error as well as the L_2. The minima where reached around the last 50 epochs of the training, the L_2 is evaluated with the model at the last epoch.

Hidden units	Validation error		L_2		Shrinkage factor	
	H	**R**	**H**	**R**	**H**	**R**
50	1.91×10^{-8}	5.05×10^{-8}	0.0015	0.0025	0.06	0.01
40	2.65×10^{-8}	1.65×10^{-8}	0.0018	0.0014	0.075	0.125
30	1.77×10^{-8}	3.40×10^{-8}	0.0015	0.0021	0.015	0.0167
20	2.50×10^{-8}	5.25×10^{-8}	0.0017	0.0027	0.1	0.25
10	5.11×10^{-8}	3.97×10^{-7}	0.0025	0.0077	0.3	0.5

two remaining hidden layers is examined in the following. For both the hydrodynamic and the rarefied regime five experiments are conducted, lowering the hidden units of the hidden layers from fifty to ten. Note that the decoder is chosen to be structurally a reflection of the encoder. Therefore only one parameter is changed. Results for **H** and **R** are shown in table A.3. Note that the contribution of over- and underfitting is negligible and therefore the training error is omitted. A model with 30 hidden units in encoder and decoder performs best with **H** and reaches a validation error of 1.77×10^{-8}. The corresponding $L_2 = 1.5 \times 10^{-3}$ with a shrinkage factor of 0.015. Overall the loss of each experiment with **H** is quiet similar an ranges from 1.77×10^{-8} to 5.11×10^{-8}. The L_2 behaves in a similar fashion and is even equal for 50 and 30 layers. A model with 40 hidden units performs best for **R**. The corresponding validation error is 1.65×10^{-8} with $L_2 = 1.4 \times 10^{-3}$, which is smaller than **H**. The shrinkage factor is 0.125. In all experiments a model with 10 hidden nodes performs worst. Training and validation error over all 4000 epochs for both experiments can be seen in fig. A.3. The aforementioned separation of training- and validation error, that was observed solely for **H**, is mitigated when moving away from 40 hidden units for encoder and decoder. Not shrinking the input in the first hidden layer only serves the performance when using **R**.

Table A.4: Results for the variation of batch sizes. Given is the minimum value of validation error as well as the the corresponding epoch. The L_2 is also given but evaluated with the model at the last epoch.

Batch Size	Validation error		L_2		Epoch	
	H	**R**	**H**	**R**	**H**	**R**
32	5.40×10^{-8}	2.17×10^{-8}	0.0024	0.0017	4998	4992
16	1.95×10^{-8}	2.06×10^{-8}	0.0015	0.0016	4999	5000
8	2.25×10^{-8}	1.03×10^{-8}	0.0017	0.0012	4965	4961
4	1.52×10^{-8}	6.30×10^{-9}	0.0013	0.0010	3956	4534
2	1.15×10^{-8}	9.18×10^{-9}	0.0012	0.0013	4956	4872

Next the mini-batch size is analysed. Epochs are increased by 1000 epochs, as training- and validation error show potential to decrease further as seen in fig. A.3 for **R** with 40 hidden nodes. Results for **H** and **R** are displayed in table A.4. Experiments are conducted with mini-batch sizes of 2, 4, 8, 16, 32. Batch sizes to the power of 2 are typically chosen as to fully exploit computational capabilities of a GPU i.e. aligning the batch size with the way memory is structured within a GPU. The smallest batch size of 2 yields the lowest validation error of 1.15×10^{-8} with corresponding $L_2 = 0.0012$ at epoch 4956 for **H**. The lowest validation error with 6.30×10^{-9} is achieved for **R** at epoch 4534 with a batch size of 4. The corresponding $L_2 = 0.001$. Compared to a batch size of 16 in the previous experiments we can observe that small batch sizes have a regularizing effect on training as described in chapter 3 and therefore are beneficial to generalization. At the same time, the lower the batch sizes are, the more unstable is the training as seen in appendix B. The oscillations which begin with batch sizes of 8 and lower, which make the training unstable, can be battled with a lower learning rate as soon as training starts to tremble. Additionally small batch sizes drastically increase training time which is why a batch size as low as 2 will not be used for the next experiments. In conclusion a batch size of 2 is omitted and for both input data a batch size of 4 is chosen. Furthermore a reduction of the learning rate from 1×10^{-4} to 1×10^{-5} is applied after the 3000th epoch.

Eight experiments with different activation functions namely ReLU, ELU, Tanh, SiLU and LeakyReLU are performed. The experiment designs and results are given in table A.5 for hidden and code activations. With **H** a combination of ELU and ELU for hidden and code activation yields the best results in validation error with 4.44×10^{-9} and a corresponding $L_2 = 0.0008$. These values are achieved at the last epoch. For **R** a combination of ReLU and ReLU for hidden and code activation produces a validation error of 7.18×10^{-9} and a corresponding $L_2 = 0.0009$. Both values are also reached close to the last epoch. Note that all models reach their lowest loss at or very close to the last epoch. The reason is the stable training after the 3000th epoch, where the learning rate is lowered to 1×10^{-5} as seen in appendix A. This measure shows in all experiments an immediate success for learning. Both validation and training error fall at the 3001st epoch and only decrease slightly thereafter. This behavior clearly shows that the updates to the free parameters θ were too big which prohibitively slowed down or even prevented the learning process. Small updates to θ made all models quickly reach a minimum. Therefore 3000 epochs or even less could have been enough to reach similar results with a lower learning rate.

If the learning rate could have been reduced whilst producing similar results remains unanswered here as the results are satisfactory. Note that for **R** in the previous experiment a validation error of 6.30×10^{-9} was achieved which is slightly lower than the current result. Nonetheless it is decided

to take the current model as the final result.

Table A.5: Results for the variation of activations for hidden-/code layers. Given is the minimum value of validation error as well as the the corresponding epoch and the L_2. The L_2 is evaluated with the models saved when the minimum validation error was achieved during training.

Activations hidden/code	Validation error		L_2		Epoch	
	H	**R**	**H**	**R**	**H**	**R**
ReLU/ReLU	9.79×10^{-9}	7.18×10^{-9}	0.0010	0.0009	5000	4998
ELU/ELU	4.44×10^{-9}	1.11×10^{-8}	0.0008	0.0012	5000	5000
Tanh/Tanh	7.83×10^{-9}	2.58×10^{-8}	0.0011	0.0018	5000	5000
SiLU/SiLU	7.69×10^{-9}	1.37×10^{-8}	0.0011	0.0013	5000	5000
LeakyReLU/LeakyReLU	1.86×10^{-8}	9.39×10^{-9}	0.0015	0.0010	5000	4997
ELU/Tanh	5.49×10^{-9}	1.87×10^{-8}	0.0008	0.0014	5000	5000
LeakyReLU/Tanh	1.00×10^{-8}	1.42×10^{-8}	0.0010	0.0012	4997	4992
ELU/SiLU	8.11×10^{-9}	1.93×10^{-8}	0.0011	0.0015	5000	5000

The final hyperparameters for both input data are summarized below in table A.6. From the initial models to the final models the decrease in validation error gained $\approx 1.5 \times 10^{-7}$ for **H** and $\approx 7.2 \times 10^{-8}$ for **R** which is 93% of the initial values for both models.

Table A.6: Summary of the final hyperparameters for both input data.

Input data	Act. hidden/code	Batch size	Width	Depth	Learning rate	Epochs
H	ELU/ELU	4	30	4	$10^{-4}/10^{-5}$	≈ 3000
R	ReLU/ReLU	4	40	4	$10^{-4}/10^{-5}$	≈ 3000

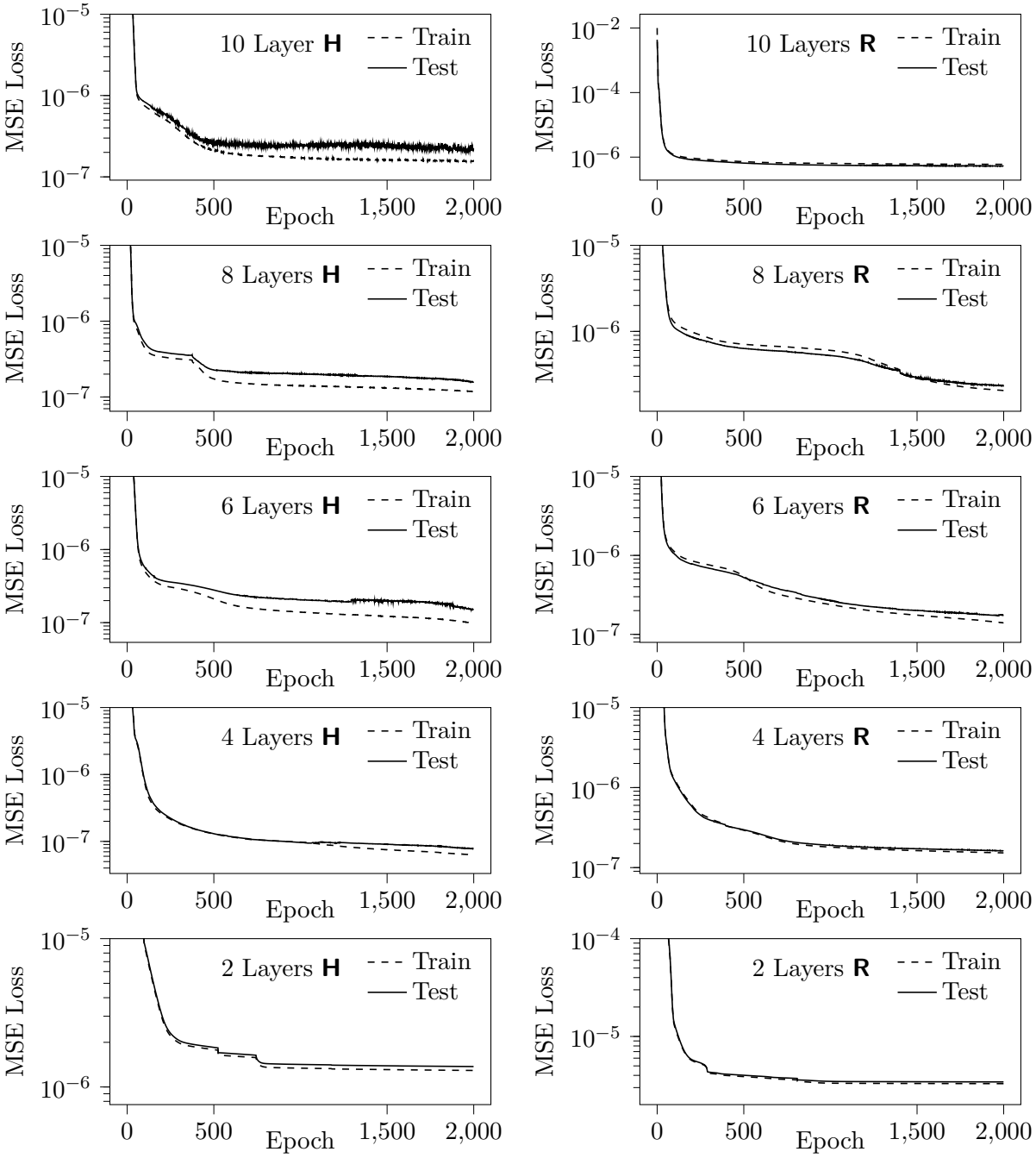


Figure A.2: Five experiments over the different depth with **H** left and **R** right. The number of layers used for every experiment are given. Training and validation loss are shown over 2000 epochs.

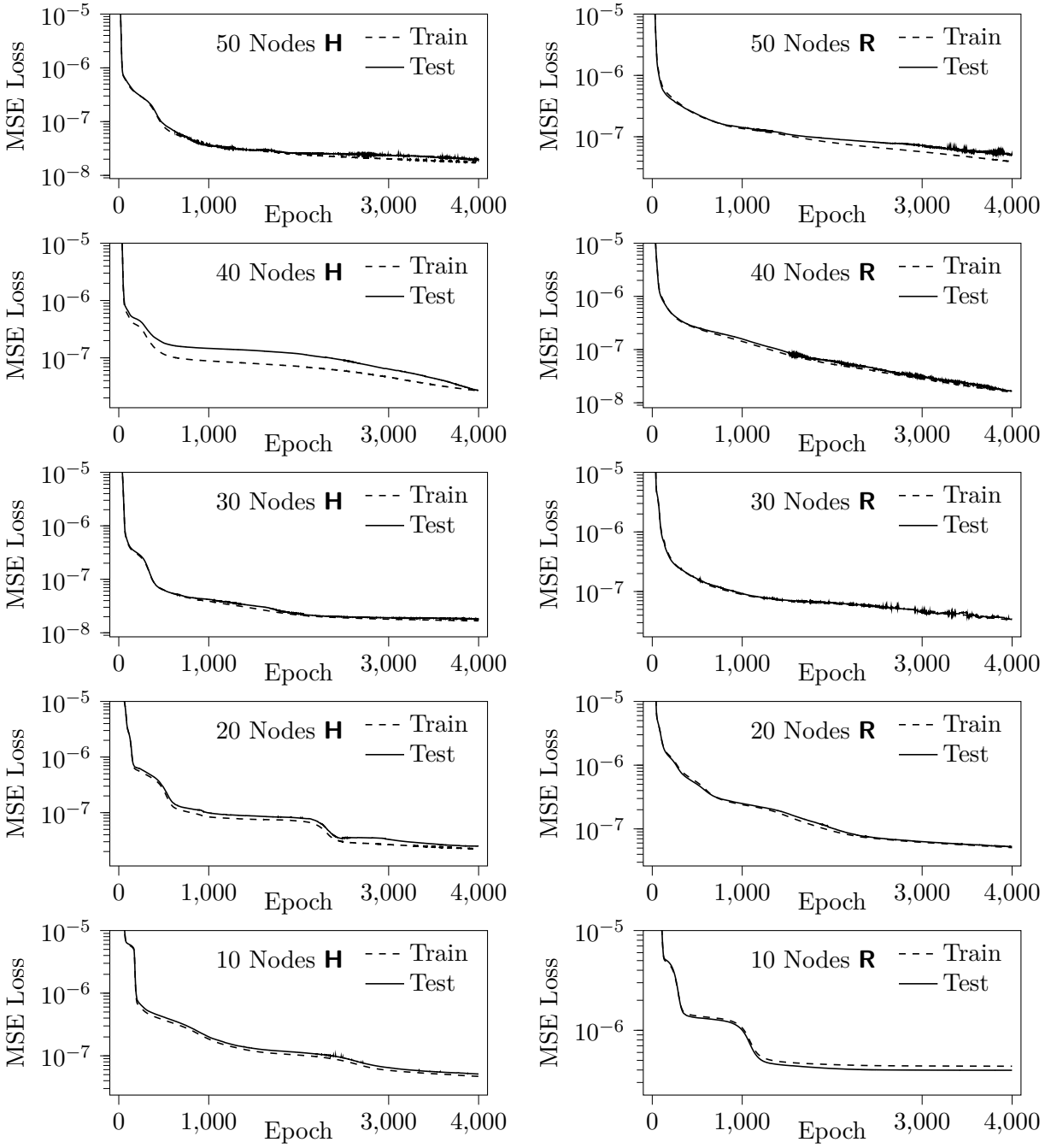


Figure A.3: Five experiments over different width with **H** left and **R** right. The number of nodes used for every experiment are given. Training and validation loss are shown over 4000 epochs.

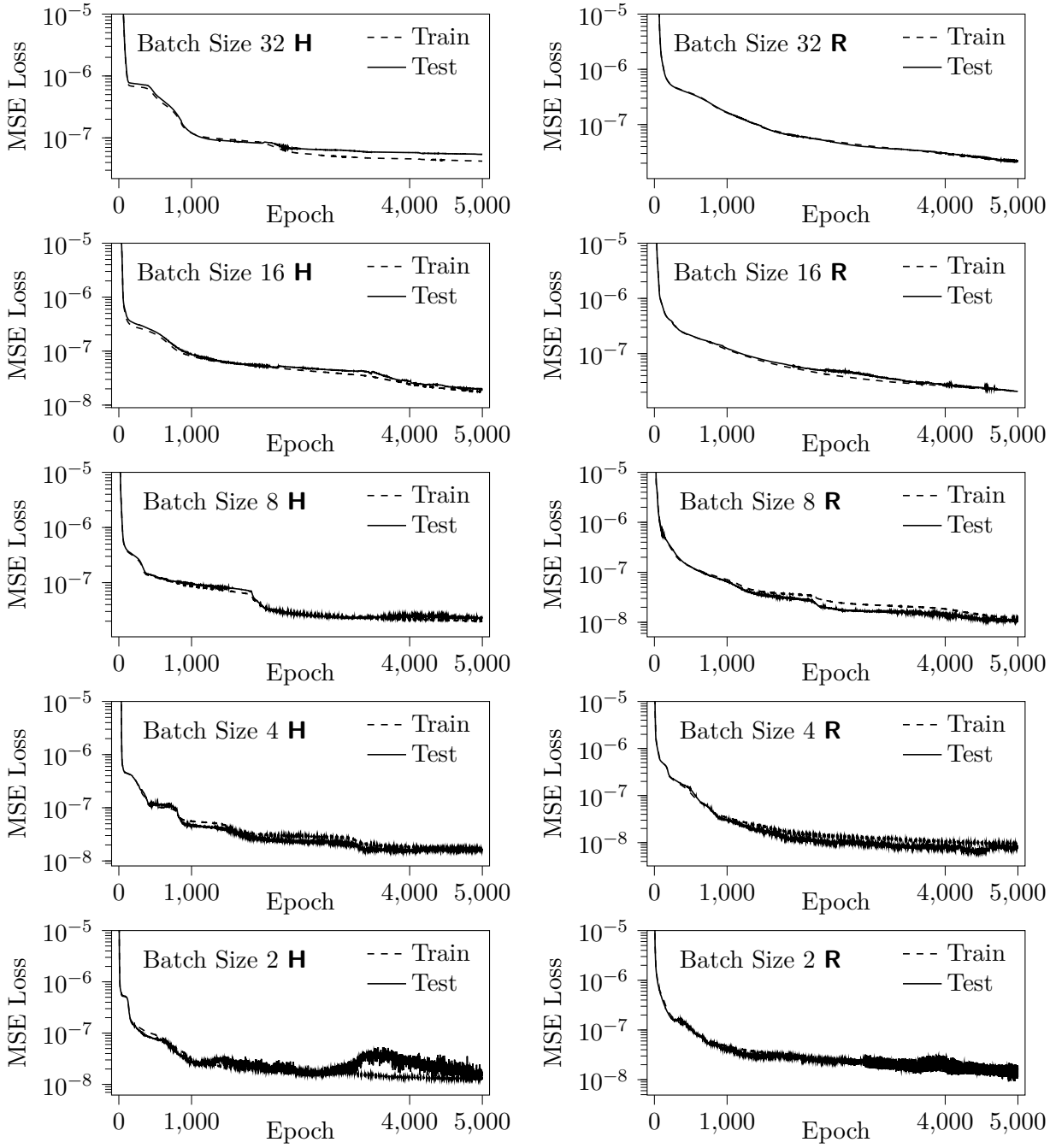
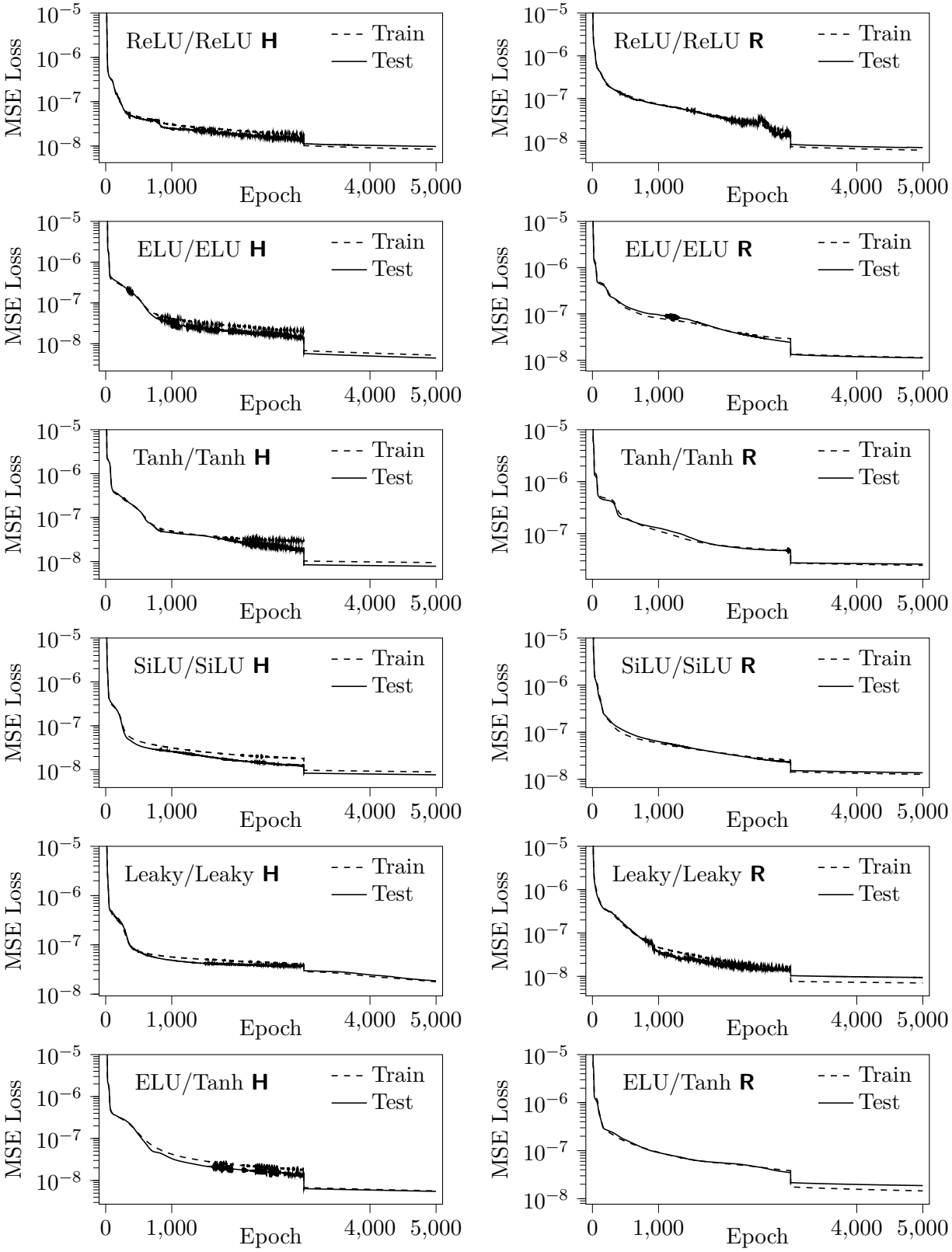


Figure A.4: Five experiments over different batch sizes with **H** left and **R** right. The batch size used for every experiment is given. Training and validation loss are shown over 5000 epochs.


ReLU/ReLU H
ReLU/ReLU R
ELU/ELU H
ELU/ELU R
Tanh/Tanh H
Tanh/Tanh R
SiLU/SiLU H
SiLU/SiLU R
Leaky/Leaky H
Leaky/Leaky R
ELU/Tanh H
ELU/Tanh R
Train
Test
MSE Loss
Epoch
0
1,000
4,000
5,000
10^{-5}
10^{-6}
10^{-7}
10^{-8}

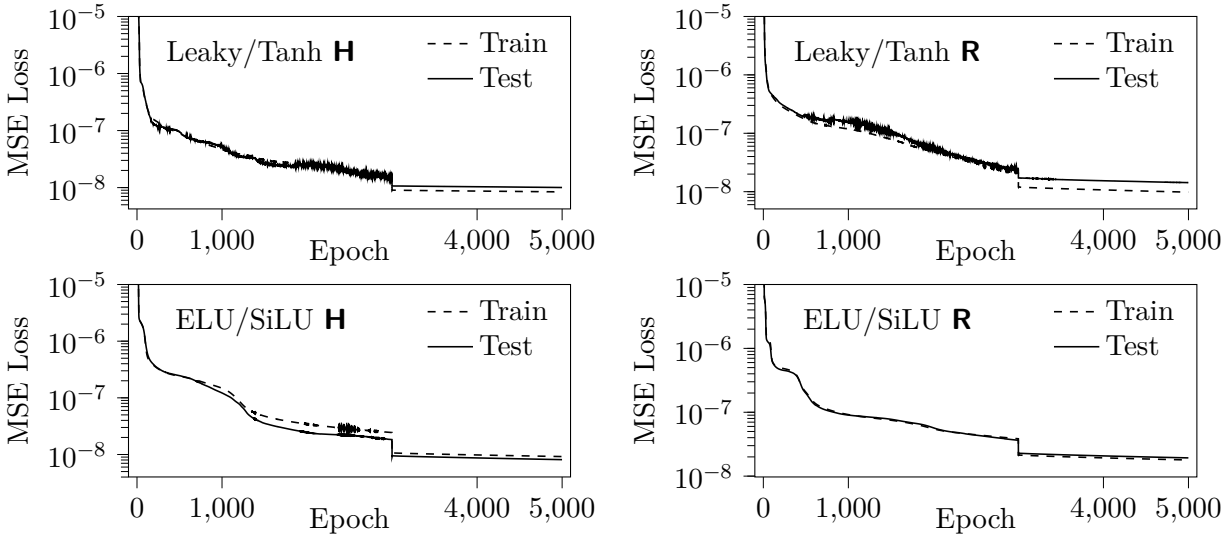


Figure A.5: Eight experiments with different combinations of activation functions for **H** (left) and **R** (right). Shown are training- and validation error over 5000 epochs.

B Hyperparameters for the Convolutional Autoencoder

The finding of appropriate hyperparameters for the convolutional autoencoder is described here. The hyperparameters include batch size, non-linear activation functions, number of epochs for training and learning rate and number of layers i.e. depth. Depth comprises kernel size, stride width as well as number of channels per layer.
This analysis follows the prior finding of hyperparameters for the fully connected autoencoder, hence utilizes insights thereof. Additionally this analysis follows a slightly different scheme than before, as there are solely 40 examples for training and validation for both rarefaction levels **H** and **R**. Hence it is tried to find a combined model that performs well on both rarefaction levels. With this measure datasets for both rarefaction levels are concatenated to one dataset of 80 examples for training and validation. Furthermore this approach gives an answer to how well a model can generalize about the BGK model for different rarefaction levels in Sod's shock tube.
Again we start with a small model which architecture and hyperparameters are summarized in table B.1. The activations and learning rate stem from conclusions made in appendix A. The application of activation functions follows the same sheme as described in appendix A seen in fig. A.1. Kernel size and stride width are chosen as to circumvent checkerboard artefacts as described in chapter 3. The number of channels is inspired by the architecture chosen in [2].

Table B.1: Initial selection of hyperparameters for training and architecture. The initial model compirses 6 layers, of which 4 are either convolutional (Conv.) or transposed convolutional (Tr.Conv.). The learning rate is switched from 10^{-4} to 10^{-4} at the 500th epoch as in appendix A. Depth counts the number of convolutional/transposed concolutional layers and excludes the fully connected layers.

Encoder				Decoder			
Layer	Type	Channels	Output size	Layer	Type	Channels	Output size
1	Conv	8	5×40	4	Lin.	-	128
2	Conv.	16	1×8	5	Tr.Conv.	16	1×8
3	Lin.	-	5	6	Tr.Conv.	8	25×200
Epochs	Act. hidden/code		Batch size	Depth	Kernel size	Stride width	Learning rate
1000	ReLU/ReLU		2	4	5×5	5×5	$10^{-4}/10^{-5}$

The number of available examples for training and validation limited as previously stated . Therefor the split in training and validation sets encompasses the risk of a bias in one of the sets. For instance, if out of the 40 examples, a few of them ,maybe 5 , contain considerable variance to the other examples and all of them are found in the training set, then the validation error could not estimate the model's ability to generalize. Therefore the k-fold algorithm described in [7] is adopted. The k-fold algorithm provides from the complete shuffled dataset, k independent splits into training

and validation set with which the model is trained. By that, each example gets the chance to be either in the training or validation set once. For an 80/20 split as used in this thesis and described in chapter 3, five independent folds can be obtained. Hence with 80 available examples the training set P_{train} consists of 74 examples and the validation set P_{val} of the remaining 16 examples. The results of said preliminary experiment are summarized in table B.2 and appendix B. The lowest

Table B.2: Training of five independent folds. Summary of minimum training- and minimum validation error for a small model with two convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached. The mean and standard deviation of the validation error are 3.55×10^{-5} and 2.13×10^{-5} respectively with a corresponding variance of 4.56×10^{-10}.

Fold	Minimum training error	Minimum validation error	L_2	Epoch
1	2.4×10^{-5}	7.0×10^{-5}	0.053	498
2	1.4×10^{-5}	1.3×10^{-5}	0.044	1000
3	1.7×10^{-5}	5.0×10^{-5}	0.057	1000
4	1.7×10^{-5}	1.8×10^{-5}	0.039	997
5	1.6×10^{-5}	2.7×10^{-5}	0.053	1000

validation error of 1.3×10^{-5} is achieved with the second fold at the last epoch. The corresponding $L2 = 0.044$. As training error is lower than validation error it is assumed, that the "difficult" examples are within the training set. Therefore the training set could be biased. The second lowest validation error of 1.8×10^{-5} and a corresponding $L_2 = 0.039$ is achieved with fold 4 at the 997th epoch. For both folds training and validation error evolve in unison and get instable even before the 500th epoch. The instability is eliminated by lowering the learning rate at said epoch, but the training does not improve thereafter as in appendix A. For the other folds a seperation of training and validation error can be observed and less instability. The mean validation error of 3.55×10^{-5} gives an estimate of the models performance with a standard deviation of 2.13×10^{-5} and variance of 4.56×10^{-10}. For the continuation of experiments fold 4 is used, as it provides a balanced split in training- and validation split which also manifests in the lowest L_2 of all folds.
Next the capacity of the model is increased by successively adding convolutional layers, thus obtaining two additional models. One encompasses three convolutional layers, the other four convolutional layers in encoder and decoder. An exact description of both architectures is summarized in table B.4. The maximum number of channels for this analysis is kept at 16. This measure decreases the growth rate of channels per layer, which is quadratic for the first model and now scales to linear in order to increase capacity rather smooth. Note that it is also possible to keep the growth rate constant while adding layers. Here this is done in the following steps.
Kernel size and stride width shrink the input as described in chapter 3. In the previous model shrinkage evolved with a pace of $\frac{1}{5}$. This reduction rate is able to reduce the input size over two succesive layers to unity. Hence the same kernel size and stride width can only be adopted by using excessive zero padding between successive convolutional layers. To overcome this issue a balance between zero padding, which cannot be omitted for four convolutional layers with the given input dimensions, and kernel size and stride width needs to be found. Here both features are chosen to have sizes 3×3. Extra padding, which is needed for one to reduce the shrinkage rate per layer in the encoder and for the other to transpose the shrinkage of the encoder in the decoder, is summarized in table B.4. In `pytorch`'s implementation of transposed convolutional layers a handy parameter can be used called output padding. This parameter provides a measure to effectively increase the

output size by one, left or at the bottom of the output, to resolve the ambiguity when stride > 1. Then the respective convolutional layer maps multiple input sizes to the same output size. Taking for example the second and sixth layer of the model with three convolutional layers in encoder and decoder. Here the convolutional layer maps a size of 23 to 8. But the respective transposed convolutional layer maps size 8 to size 22. In this case output padding resolves the issue by increasing the output size by one.

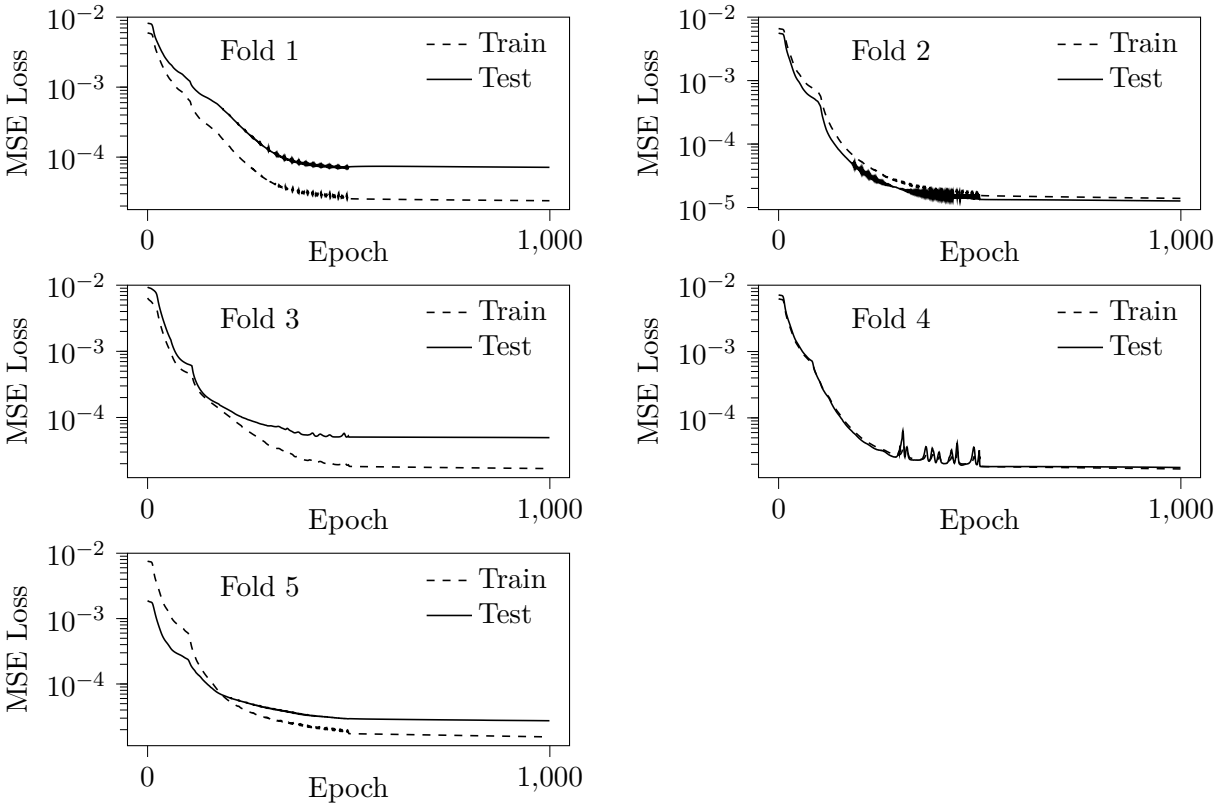


Figure B.1: Training- and validation error over 1000 epochs for five independent folds.

Results for both models are summarized in table B.4. Training- and validation error for the model with three convolutinal layers achieve slightly better results as the small model. On the contrary does the L_2 not achieve the resuls of the small model with $L_2 = 0.044$. In appendix B the evolution of training- and validation error is depicted over 2000 epochs. The model with three convolutional layers shows an unstable training after the 500th epoch, which is ignored for now. It can be observed that the model with four convolutional layers is underfitting and does not achieve comparable results to the small model.

Next the batch size is increased from two to four. The initial batch size was chosen so small as the limited available data permits reasonable training time. The results for all three models are

Table B.3: Architecture for the models with three (left) and four (right) convolutional layers in encoder and decoder. Kernel size and stride width are 3×3 for both features and models.

Encoder				
Layer	Type	Channels	Padding	Output
1	Conv	4	in: 1/1	9×67
2	Conv.	8	in: 0/1	3×23
3	Conv.	16	in: 0/1	1×8
4	Lin.	-	-	5

Decoder				
Layer	Type	Channels	Padding	Output
5	Lin.	-	-	128
6	Tr.Conv.	16	in: 0/1 out: 0/1	1×8
7	Tr.Conv.	8	in: 0/1 out: 0/1	3×23
8	Tr.Conv.	1	in: 0/1	25×200

Encoder				
Layer	Type	Channels	Padding	Out
1	Conv	2	in: 3/2	10×68
2	Conv.	4	in: 3/2	5×24
3	Conv.	8	in: 3/2	3×9
4	Conv.	16	in: 3/3	1×3
5	Lin.	-		5

Decoder				
Layer	Type	Channels	Padding	Out
6	Lin.	-	-	128
7	Tr.Conv.	16	-	3×9
8	Tr.Conv.	8	in: 2/2 out: 0/1	5×24
9	Tr.Conv.	4	in: 3/2	9×68
10	Tr.Conv.	1	in: 1/2	25×200

Table B.4: Results for increasing number of layers. Summary of minimum training- and minimum validation error for a model with three and a model with convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached.

Layer	Minimum training error	Minimum validation error	L_2	Epoch
3	1.4×10^{-5}	1.5×10^{-5}	0.044	1975
4	7.3×10^{-4}	8.1×10^{-4}	0.327	1995

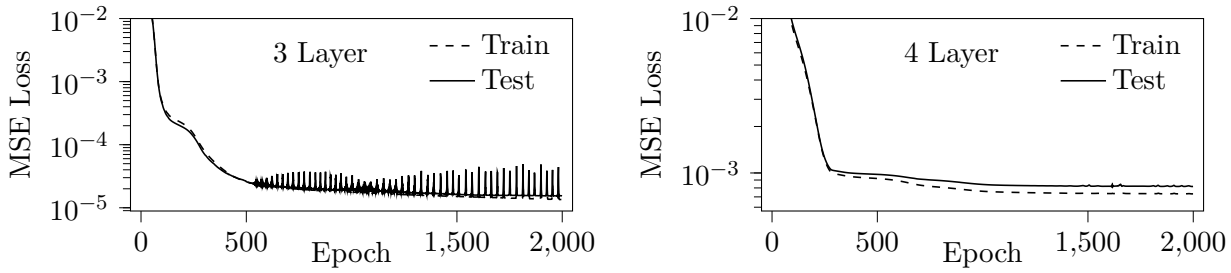


Figure B.2: Increasing the number of layers. Shown are training- and validation error over 2000 epochs a model with two and a model with three convolutional layers in the encoder. Channels are chosen to reach a maximum of 16 in the last layer of encoder. Kernel size and stride width are both 3×3.

shown in table B.5. The models with two and three convolutional layers benefit from increasing the batch size. The small model reaches the best training- and validation error with 6.0×10^{-6} and 8.0×10^{-6} respectively and $L_2 = 0.03$. In appendix B it can be observed, that the increased batch size results in a stable training for both shallow models. The underfitting of the deep model with four convolutional layers in encoder and decoder is increased.

Table B.5: Increasing the batch size to four. Summary of minimum training- and minimum validation error for the models with two, three and four convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached.

Layer	Min. training error	Min. validation error	L_2	Epoch
2	6.0×10^{-6}	8.0×10^{-6}	0.030	1999
3	1.0×10^{-5}	1.3×10^{-5}	0.038	1965
4	6.0×10^{-3}	6.9×10^{-3}	0.94	109

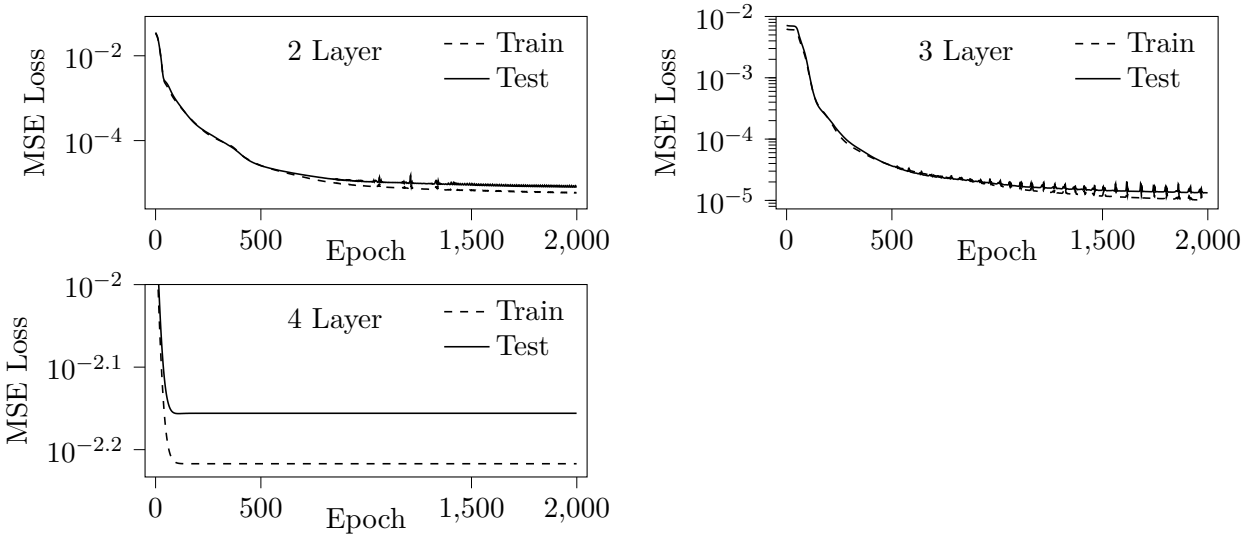


Figure B.3: Increasing the batch size to four. Shown are training- and validation error over 2000 epochs for the models with two, three and four convolutional layers in the encoder.

The underfitting of the deepest model is now tackled with increasing te capacity of the model by raising the channel growth rate to quadratic, as it is found in [2]. The same is performed for the model with three convolutional layers in encoder and decoder. Additionally the channel sizes of the small model are increased. The results and available channel sizes in order of appearance in the encoder are summarized in table B.6. Note that the decoder mirrors the channel sizes of the encoder. The small model does not learn at all as seen in appendix B. The training was repeated several times, but no improvement could be produced. The reason for this behavior is not known and can't be classified as underfitting and overfitting, as there is no change in training- and

validation error observed. On the other hand does the deep network with four convolutional layers achieve comparable results to the small model in the previous experiment. The model with three convolutional layers improves slightly. In conclusion, the quadratic growth rate of the channels starting from eight channels is suitable for the following experiments, for which the three layer model is discarded as the other models reach the lowest validation error so far.

Table B.6: Increasing the channel growth rate to quadratic. Summary of minimum training- and minimum validation error for the models with two, three and four convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached. Note that the channel sizes of the two layer model are only increased, the growth rate has already been quadratic.

Layer	Channels	Minimum training error	Minimum validation error	L_2	Epoch
2	16,32	6.8×10^{-3}	7.6×10^{-3}	1.0	0
3	8,16,32	9.0×10^{-6}	1.1×10^{-5}	0.037	1985
4	8,16,32,64	7.0×10^{-6}	9.0×10^{-6}	0.033	1953

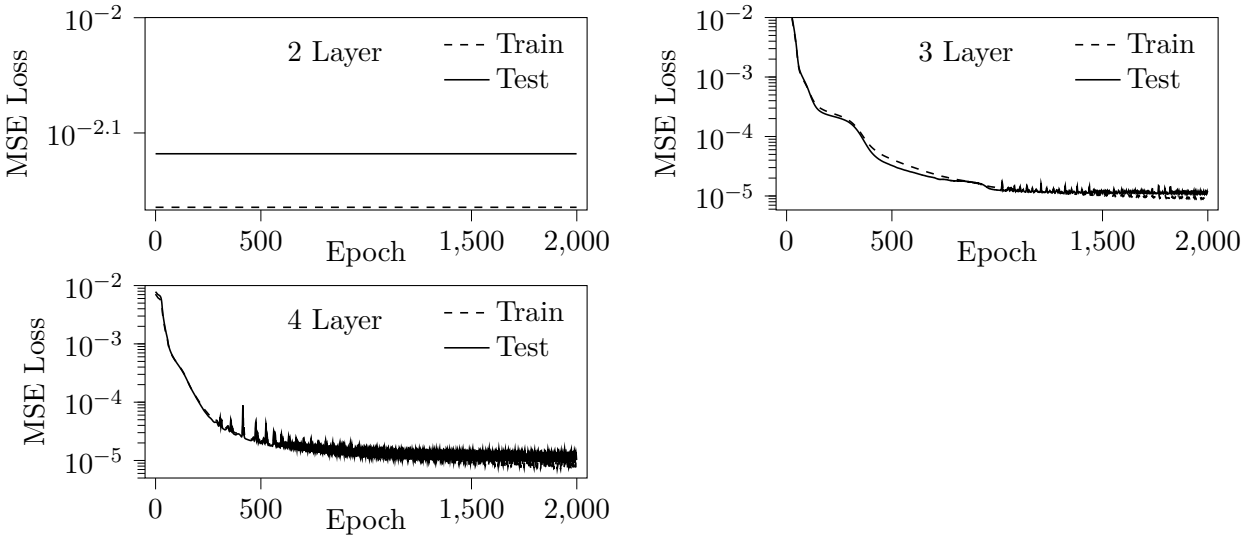


Figure B.4: Increasing the channel growth rate to quadratic. Shown are training- and validation error over 2000 epochs for the model with two, three and four convolutional layers in the encoder.

The variation of activation functions follows the same scheme as already outlined in appendix A. Eight experiments are conducted using ReLU, ELU, Tanh, SiLU and LeakyReLU in different combinations for the hidden convolutional layers and the code layer. Results are summarized in table B.7. Training- and validation error over 2000 epochs are displayed in fig. B.8. For both models the validation error reaches the region of 10^{-6} using combinations of ELU, SiLU and Tanh. With applying rectifiers, the validation error stays in the region of 10^{-5}. Especially, with utilizing SiLU/SiLU for the four layer model, the validation error reaches a minimum of 6.0×10^{-6}. Utilizing the combinations ELU/SiLU and ELU/Tanh for the two layer model also yields a minimum validation error

of 6.0×10^{-6}. In addition, this combination produces the smallest L_2 with $L_2 = 0.024$ applying ELU/SiLU for the four layer model and applying ELU/Tanh for the two layer model produces $L_2 = 0.023$. These values correspond to the minimum training error which is 4.0×10^{-6} and 3.0×10^{-6} respectively. Here the four layer model overfits after the 500th epoch which produces, compared to the other models, a significant generalization gap. The same logic applies to the two layer model with ELU/Tanh, but yields a smaller generalization gap. In general, specifically the training of the four layer model is instable for all combinations of activation functions. The greater amount of free parameters, compared to the two layer model, requires a smaller learning rate or a greater batch size to stabilize the training. Finally, the combination of SiLU/SiLU is chosen for the four layer model and ELU/SiLU for the two layer model. Both choices lead to the lowest validation error, while maintaining the generalization gap small.

A comparison with the validation loss of the fully connected model suggests, that the two- and

Table B.7: Variation of activations for hidden-/code layers. Summary of minimum training- and minimum validation error for the models with two and four convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached.

Act. hid./code	Min. training error		Min. validation error		L_2		Epoch	
	2 Layer	4 Layer	2 Layer	4 Layer	2 Layer	4 Layer		
ELU/ELU	5.0×10^{-6}	4.0×10^{-6}	7.0×10^{-6}	7.0×10^{-6}	0.026	0.031	1969	1365
ELU/SiLU	5.0×10^{-6}	4.0×10^{-6}	6.0×10^{-6}	8.0×10^{-6}	0.026	0.024	1991	1808
ELU/Tanh	3.0×10^{-6}	5.0×10^{-6}	6.0×10^{-6}	9.0×10^{-6}	0.023	0.029	1998	1498
Leaky/Leaky	6.0×10^{-6}	7.0×10^{-6}	1.1×10^{-5}	1.0×10^{-5}	0.032	0.032	1976	1971
Leaky/Tanh	5.0×10^{-6}	6.0×10^{-6}	7.0×10^{-6}	9.0×10^{-6}	0.030	0.035	1977	1722
ReLU/ReLU	8.0×10^{-6}	7.0×10^{-6}	1.3×10^{-5}	1.1×10^{-5}	0.036	0.036	1984	1989
SiLU/SiLU	6.0×10^{-6}	6.0×10^{-6}	8.0×10^{-6}	6.0×10^{-6}	0.030	0.035	1972	1550
Tanh/Tanh	8.0×10^{-6}	5.0×10^{-6}	8.0×10^{-6}	8.0×10^{-6}	0.033	0.030	1999	975

three layer model gets stuck at a local minimum of $\min J(\theta)$. Therefore the input data is augmentated for the following experiments. All examples in the training- and validation set are rotated around their center by 180° and flipped about their central vertical axes. These methods add examples the input data without altering the information about the flow field present in each example. In fig. B.5 is an example in its original, rotated and flipped version. The flow field stays the same, just the direction in which it evlves has been altered. Using this method, available examples for training and testing triple from 80 to 240 examples.

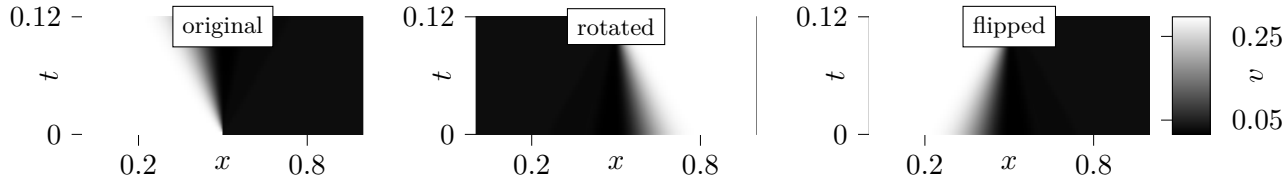


Figure B.5: Rotated and flipped version of one original example from the dataset showing v over x and t.

Results for using augmented data are summarized in table B.8. Two experiments are conducted for each model. One with learning rate adjustment, which decreases from 1×10^{-4} to 1×10^{-5} after the 1250th epoch, and one without. The minimum validation error ranges between 2.2×10^{-5} and 1.4×10^{-5}. Therefore data augmentation could not decrease the validation error further. Instead it increases. But, as seen in fig. B.6 and fig. B.7, the training is stabilized compared to the previous experiments. In addition overfitting cannot be obeserved for the two layer model. The four layer model overfits after the around the 1150th epoch. Furthermore both models benefit from reducing the learning rate. A drop in training- and validation error can be observed after the 1250th epoch for both models.

Table B.8: Data augmentation with and without learning rate adjustement after the 1250th epoch. Summary of minimum training- and minimum validation error for the models with two and four convolutional layers in the encoder as well as the corresponding L_2 and the epoch in which those values are reached.

Layer	Min. training error	Min. validation error	L_2	Epoch
2	1.5×10^{-5}	1.4×10^{-5}	0.048	2000
2 (lr adjusted)	2.2×10^{-5}	2.2×10^{-5}	0.064	2000
4	7.0×10^{-6}	1.6×10^{-5}	0.034	1991
4 (lr adjusted)	1.1×10^{-5}	1.9×10^{-5}	0.045	2000

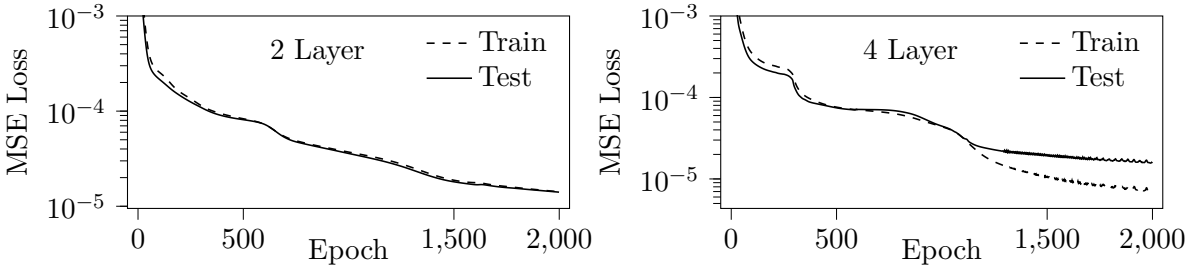


Figure B.6: Using augumented data. Shown are training- and validation error over 2000 epochs for the model with two and four convolutional layers in the encoder.

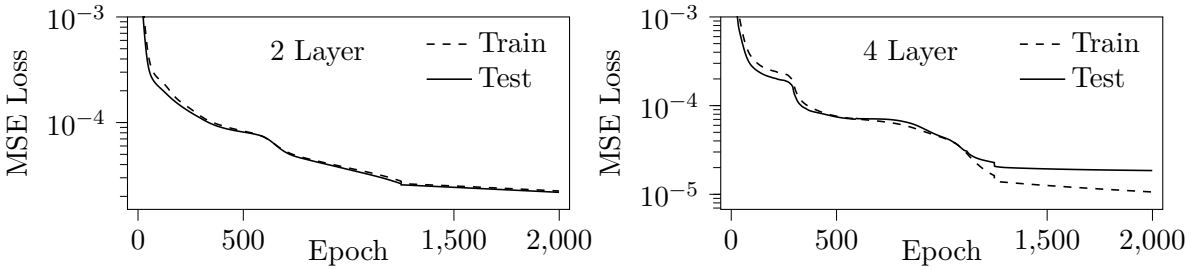
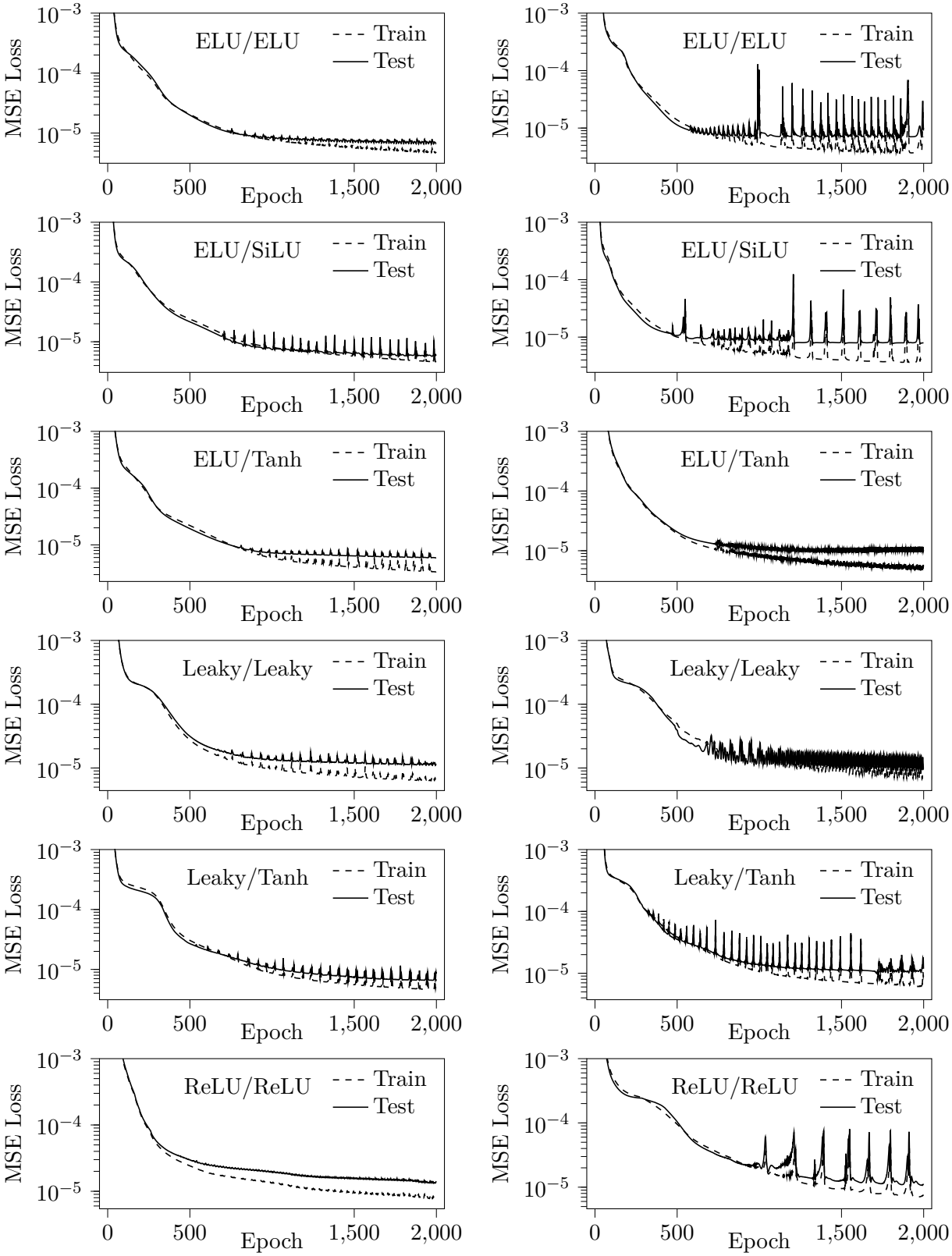


Figure B.7: Learning rate adjustment with augmented data. Shown are training- and validation error over 2000 epochs for the model with two and four convolutional layers in the encoder.


ELU/ELU
ELU/SiLU
ELU/Tanh
Leaky/Leaky
Leaky/Tanh
ReLU/ReLU
Train
Test
MSE Loss
Epoch
10^{-3}
10^{-4}
10^{-5}
0
500
1,500
2,000

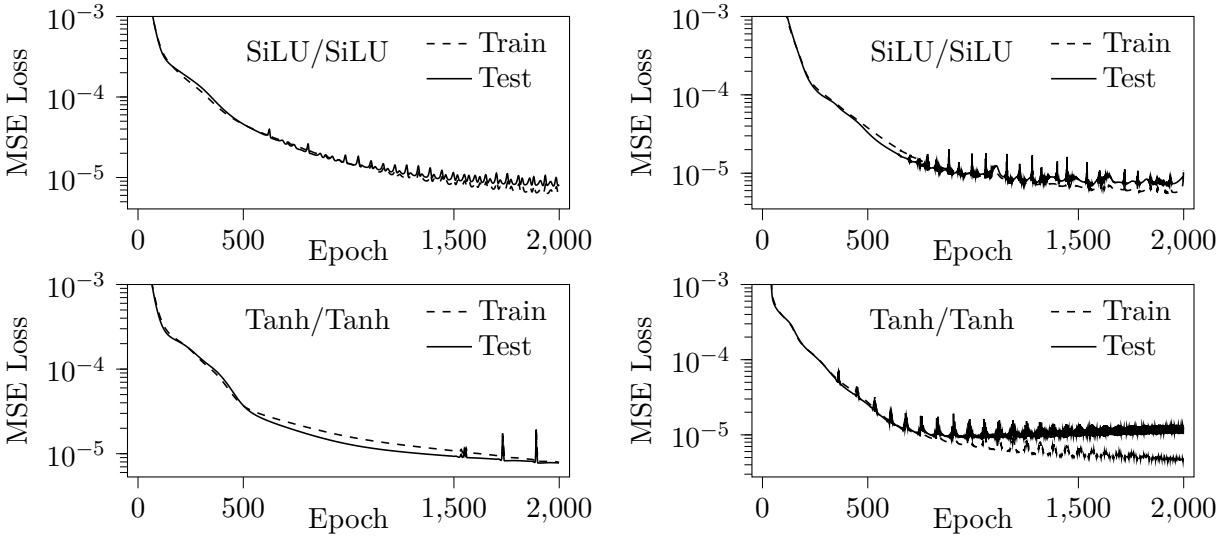


Figure B.8: Eight experiments with different combinations of activation functions for the model with two (left) and four (right) convolutional layers in encoder. Shown are training- and validation error over 2000 epochs.

To this end the finding of hyperparameters for a convolutional autoencoder using both rarefaction levels as input data failed to produce comparable results to those obtained by the fully connected autoencoders using both rarefaction levels as separate datasets. The analysis stops here out of brevity, but is far from an end. Changing the kernel size and especially the stride width, which yields non overlapping kernel positions in all models, changing the loss function to for example `pyTorch`'s BCEWithLogitsLoss or even adding more layers and so forth are still not analyzed, but are proposed for further investigations. As a final model the two layer model trained without data augmentation is chosen. This model encompasses the smallest number of free parameters while achieving the lowest validation error.

Table B.9: Final model.

Layer	Channels	Activations	Batch size	Learning rate	Num. Epochs
2	8,16	ELU/SiLU	4	1×10^{-4}	2000

Bibliography

[1] Bhatnagar, Gross and Krook, *A model for collision processes in gases*, .

[2] K. Lee and K. T. Carlberg, *Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders*, .

[3] S. R. Bukka, R. Gupta, A. R. Magee and R. K. Jaiman, *Assessment of unsteady flow predictions using hybrid deep learning based reduced order models*, 2020.

[4] T. Franz, *Reduced-order modeling of steady transonic flows via manifold learning.* 2016.

[5] S. L. Brunton and J. N. Kutz, *Data driven science and engineering.* 2019.

[6] F. Bernard, A. Iollo and S. Riffaud, *Reduced-order model for the bgk equation based on pod and optimal transport*, .

[7] I. J. Goodfellow, Y. Bengio and A. Courville, *Deep Learning.* MIT Press, Cambridge, MA, USA, 2016.

[8] D. Rumelhart, G. Hinton and R. Williams, *Learning internal representations by error propagation*, .

[9] D. H. Ballard, *Modular learning in neural networks*, .

[10] S. Rifai, P. Vincent, X. Muller, X. Glorot and Y. Bengio, *Contractive auto-encoders: Explicit invariance during feature extraction*, .

[11] S. Rifai, Y. N. Dauphin, P. Vincent, Y. Bengio and X. Muller, *The Manifold Tangent Classifier.* Curran Associates, Inc., 2011.

[12] S. Rifai, Y. Bengio, Y. Dauphin and P. Vincent, *A generative process for sampling contractive auto-encoders*, 1206.6434.

[13] S. A. Schaaf, *Mechanics of Rarefied Gases*, *Handbuch der Physik* **3** (Jan., 1963) 591–624.

[14] G. Puppo, *Kinetic models of bgk type and their numerical integration*, 1902.08311.

[15] G. A. Sod, *Review. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws*, *Journal of Computational Physics* **27** (Apr., 1978) 1–31.

[16] J. Reiss, *Skript zu cfd 1*, 1603.07285.

[17] Y. LeCun, L. Bottou, G. Orr and K. Müller, *Efficient backprop*, in *Neural Networks: Tricks of the Trade*, 2012.

[18] K. He, X. Zhang, S. Ren and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, 1502.01852.

[19] "Pytroch documentation." https://pytorch.org/docs/stable/index.html, 2019.

[20] V. Bushaev, "Stochastic gradient descent with momentum." https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d, Dec, 2017.

[21] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 1412.6980.

[22] T. Lane, *Transposed convolutions explained with ms excel!*, Nov, 2018.

[23] M. Divyanshu, *Transposed convolution demystified*, Mar, 2020.

[24] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, 1603.07285.

[25] M. Ohlberger and S. Rave, *Reduced basis methods: Success, limitations and future challenges*, 1511.02021.

[26] J. Koellermeier, Y. Fan, M. Rominger and G. Samaey, "Moment models for kinetic equations." 2020.

[27] F. Fan, J. Xiong, M. Li and G. Wang, *On interpretability of artificial neural networks: A survey*, 2021.

Acknowledgement

Hilfsmittel

.

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen keine Einwände, die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Berlin, den 28. Mai 2021

Zachary Schellin