Model Order Reduction of Rarefied Gases using Neuronal 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: *28. Mai 2021*

Abstract

Abstract here

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 operates in model order reduction for data visualization/analysis since the 80's (Quelle?) and has now surfaced in fluid mechanics. This thesis will cover the use of artificial intelligence for model order reduction in fluid mechanics.

1.1 Objective of this thesis

Due to the non-linearity of transport problems in particular shock fronts, the construction of a robust ROM for those cases poses several challenges.

1.2 Thesis outline

1. What is the BGK Model
2. What is the SOD and BGK in SOD
3. What is deep learning what are autoencoders
4. hyperparameters for autoencoders
5. What is a reduced order model
6. offline phase
7. my data in 1d bgk in sod shock tube FOM Data
8. what is pod and Rb by POD
9. RB by autoencoder -online phase
10. what is my ROM
11. results - results by ROM for FC and for CONV

12. variation of intrinsic variables

13. Comparion to POD intrinsic variables number and quality

1.3 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)[2][3]. 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.

$$f(x) \approx \tilde{f}(x) \quad \text{with} \tilde{f} = \sum_{k=1}^{r} a_k \psi_k(x) \qquad \text{where} \qquad \psi_k \quad \text{are orthonormal functions.} \tag{1.1}$$

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 $\sim$ 1 % in [4]. 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[5] 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"[6, 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) [7]. 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 [8]. The so called bottleneck of autoencoders yields a non-smooth and entangled representation thus beeing uninterpretable by practitioners[9] 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 [9]. Subsequent development emerges with the manifold tangent classifier (MTC) [10]. 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 [11].... 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 kinetic gas model, the BGK model, on which a model order reduction will be performed in the following. In addition the SOD-shocktube, on which the BGK model will be tested is discussed.
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 [4]. 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 [12].

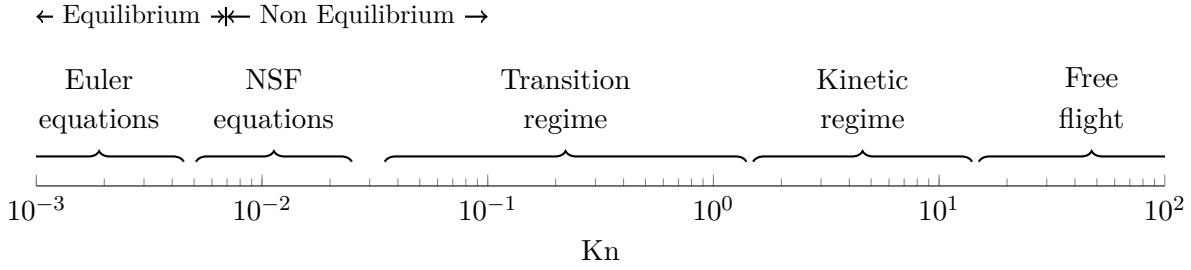


Figure 2.1: Partitioning of Kn, the Knudsen number, into levels of rarefaction. The Euler equations can be used up to approximately $Kn < 0.01$ and describe a „continuum flow“. A “slip flow“ can be defined in the interval $0.01 < Kn < 0.1$, termed as slightly rarefied in [12]. Here the Navier-stokes-Fourrier 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 [13].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 [4]. 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 [13]. 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 spacewhich can be obtained with

$$\rho(x,t) = \int f \, \mathrm{d}v, \tag{2.6}$$

$$\rho(x,t)u(x,t) = \int vf \, \mathrm{d}v, \tag{2.7}$$

$$E(x,t) = \int \frac{1}{2}v^2 f \, \mathrm{d}v, \tag{2.8}$$

as explained in [13]. 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).

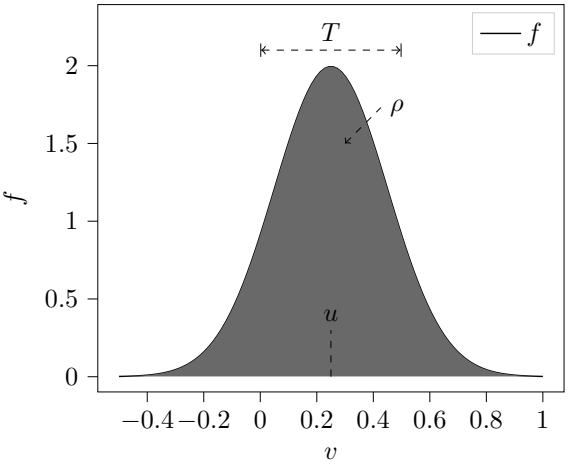


Figure 2.2: Illustration of the linkage between the macroscopic quantities of the gas flow and the distribution function f.

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 [13]. These are

$$\partial_t \rho + \partial_x(\rho u) = 0, \tag{2.9}$$

$$\partial_t(\rho u) + \partial_x(\rho u^2 + p) = 0, \tag{2.10}$$

$$\partial_t E + \partial_x(u(E + p)) = 0, \tag{2.11}$$

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

$$< \Phi(v) f > - < \Phi(v) M_f >= 0, \tag{2.12}$$

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

$$\sum_k w_k \Phi(v_k)[f(t, v_k, x) - \exp(\alpha(x,t)\Phi(v_k))] = 0. \tag{2.13}$$

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. (2.9), eq. (2.10) and eq. (2.11) found in [13].

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 [14]. 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 [15].

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 [14].

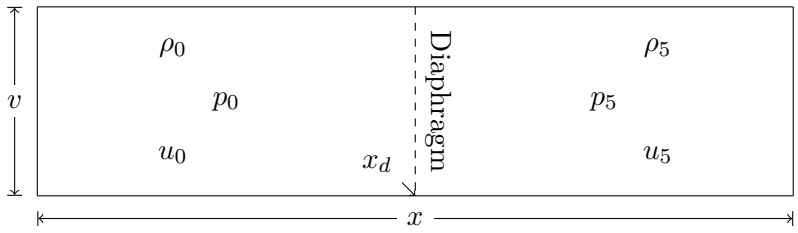
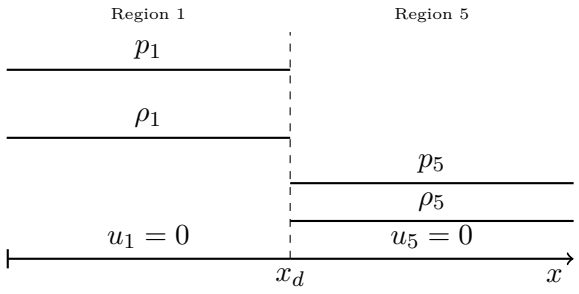
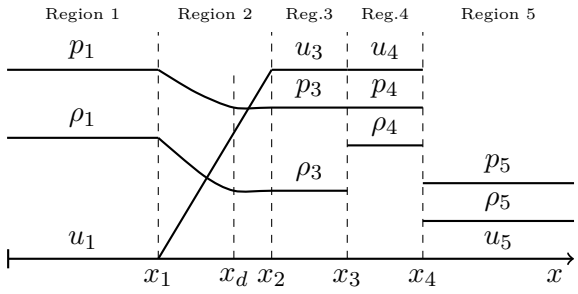


Figure 2.3: Problem setup for Sod's shock tube in 1D translated for the BGK model in velocity v and space x. A diaphragm is positioned at x_d, seperating the whole domain in two regions (region 1 and region 5). Initial conditions for density ρ, pressure p and macroscopic velocity u are indicated.

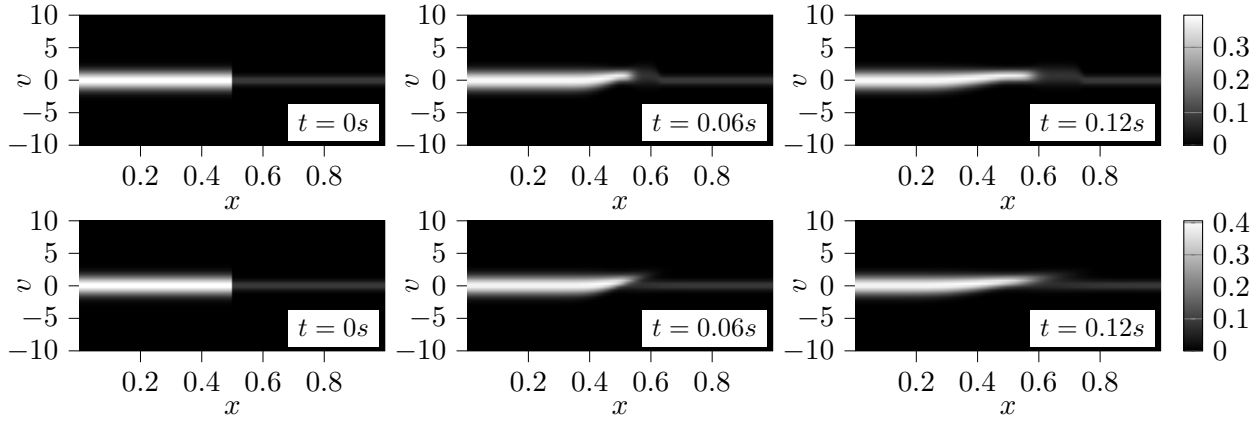
At $t > 0$ the diaphragm is broken, which leads to the formation of five regions which are depicted in fig. 2.4b. Between x_1 and x_2 we find the head and tail of the rarefaction wave traveling left. The solution for ρ, p and u is continuous in this area. The rarefaction wave is clearly discernible for 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 [14] 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.



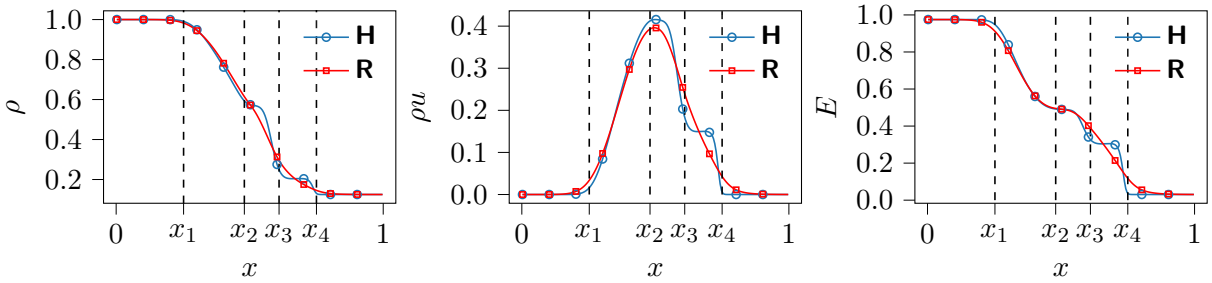
(a) Sod's shock tube at $t = 0$. The whole domain is split into two regions with corresponding initial conditions for pressure p, density ρ and macroscopic velocity u. Position of the diaphragm is labeled as x_d.



(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 [3]. 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 n of basis modes to reconstruct the dynamics.
An optimal basis can be provided by POD. It leverages a physically interpretable spatio-temporal decomposition of the input data [3]. 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 [3]. 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 [3], 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 [3].
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 [6]. 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

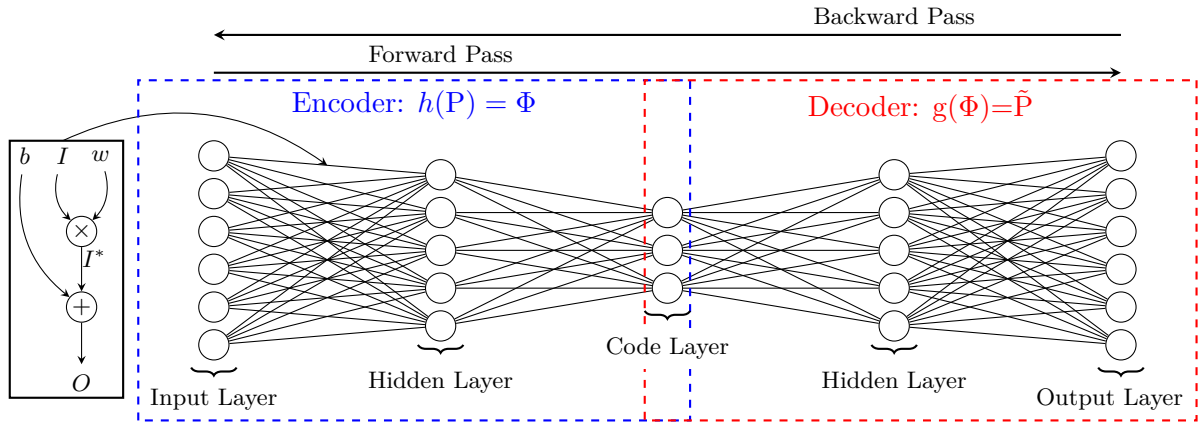


Figure 3.1: Scheme of an undercomplete autoencoder with five fully connected layers. Input layer, output layer, the code or bottleneck layer and two hidden layers. Every layer is made up of nodes, which are represented as circles. The more hidden layers are added to the architecture the deeper is the 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 [6].

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 [3] 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 [6] for an overview, [16] for the default initialization in `pytorch`, which is also used in this thesis and [17] 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 [18] 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 [16]. 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 [6][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 [6][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 [19]. 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 [20]. 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 [20], 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 [6] 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 [6][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

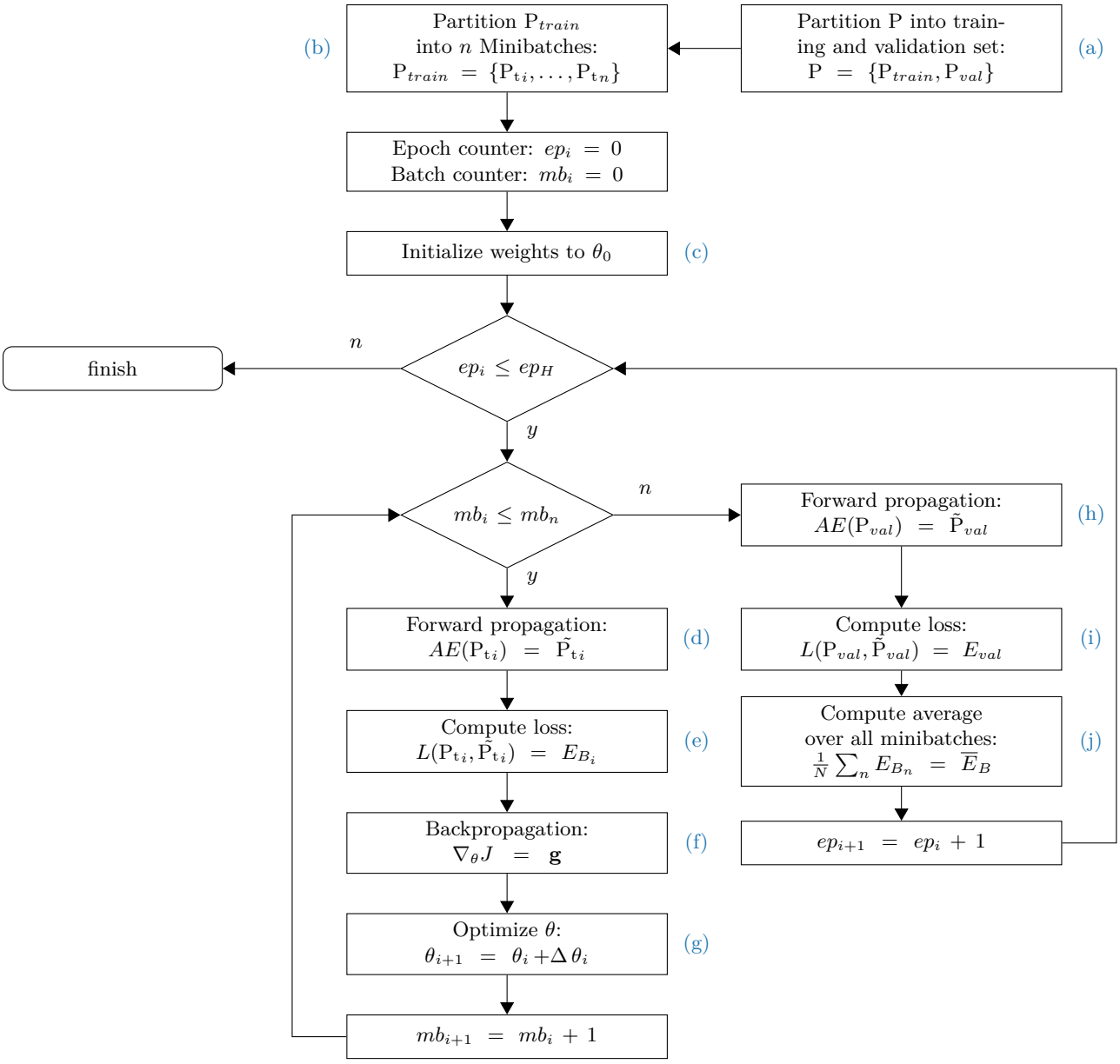


Figure 3.3: Flowchart of the training process used in this thesis for an autoencoder. The data set P for training is split up into a training P_{train} and a validation P_{val} set with a 80/20 ratio (a). The P_{train} set is equally devided into n minibatches (b). The network will be trained over H epochs and n minibatches. First the weights w are initialized as in [16] 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.

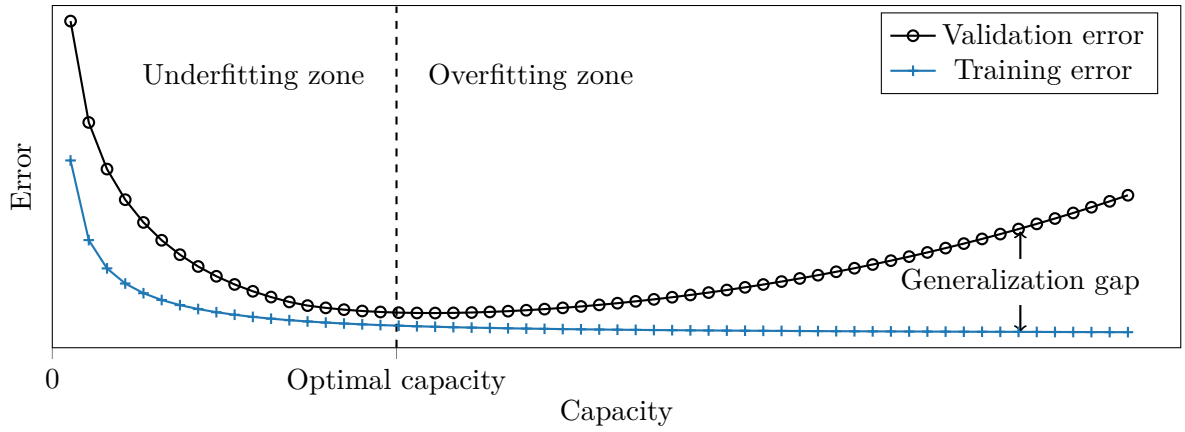


Figure 3.4: Figurative example of how capacity influences the evolution of training and validation error. With increasing capacity the model i.e. autoencoder is able to fit the training and validation set thus training and validation error decreases. Typically the training error is less than the validation 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.

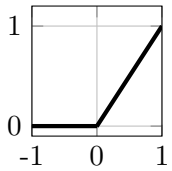
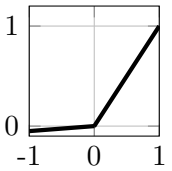
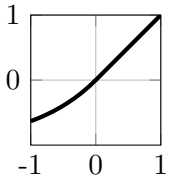
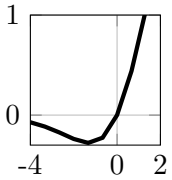
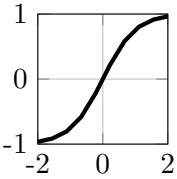
transformations as in eq. (3.7). Thus the networks so called hypothesis space solely includes linear transformations of the input. The hypothesis space from which a neural network can choose the composition of the solution can be enriched with non-linear functions by simply evaluating layers through 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`[18]. 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

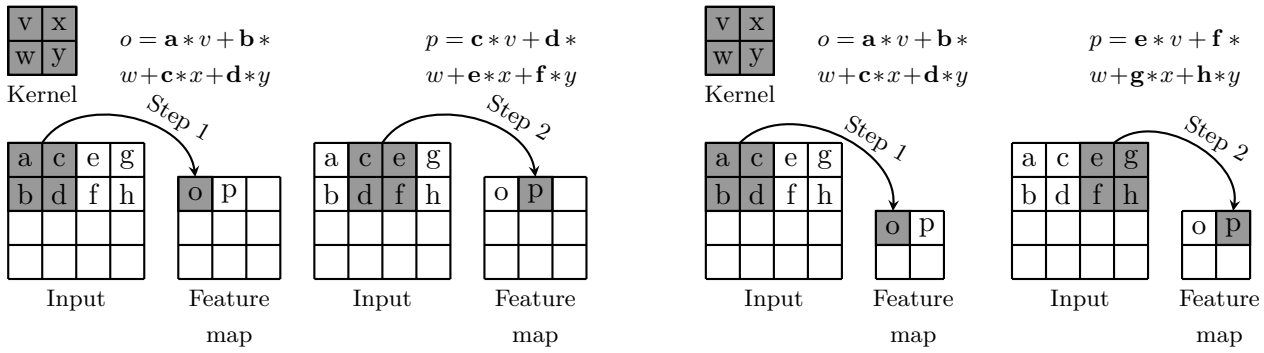
Table 3.1: Non-linear activation functions of type rectifier are the rectified linear unit (ReLU), it's leaky variant LeakyReLU, the exponential version ELU. The negaive slope of LeakyReLU below zero α is typically, and also in this thesis set to $\alpha = 0.001$. Activations based on tangens hyperbolicus (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. 3.7. 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. 3.7 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 [6] [**?**]. Details on why the cross correlation is used instead of the convolution is discussed further in [6]. 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. 3.7 (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, 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

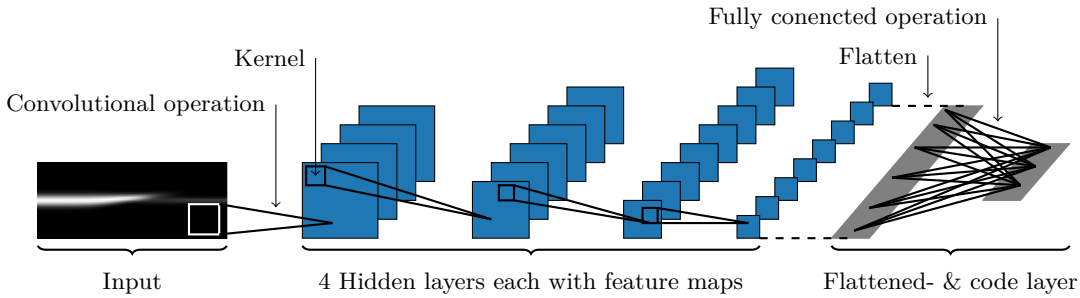
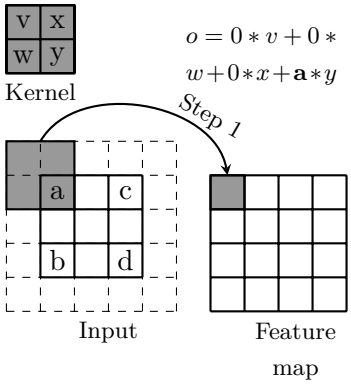
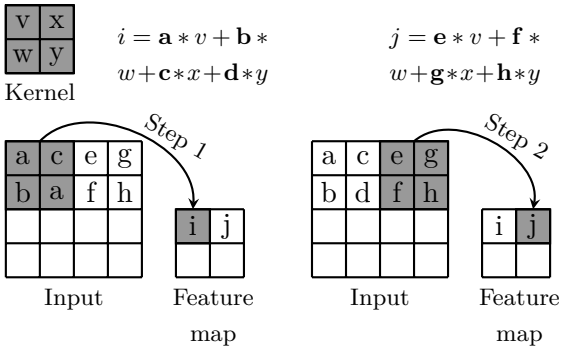


Figure 3.6: Schematic representation of typical components and their structural set-up in a convolutional neural network (CNN). Shown is the convolutional operation and the associated kernel, the flattening and successive fully connected operation. First a kernel moves over the one dimensional input 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.

and encoder in autoencoders. Convolutional layers on the other hand are not able to upsample a given input. Therefore transposed convolutional layers are applied called `ConvTranspose1d`, `ConvTranspose2d` and `ConvTranspose3d` in `pyTroch`.



(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.

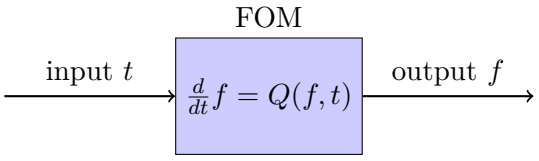


(b) Two dimensional example of of a **two** strided convolution over a 4×4 input matrix with a 2×2 kernel matrix. The equations for obtaining the components i and j of the feature map are given. Note that all biases are omitted for simplicity and bold symbols should help readability and do not represent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

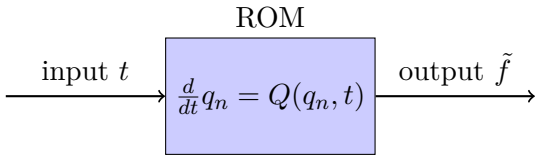
Figure 3.7: Comparison of a one strided convolution (a) and a two strided convolution (b). Illustrated are two steps of a kernel matrix moving over an input matrix. The two strided convolution yields an increased downscaling compared to the one strided convolution.

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 [4][5][23]. 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, with o beeing the number of elements in v and p in n. With $p << o$ we obtain a reduced order model (ROM) $q(x, n, t)$ 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, though we discuss the 1D case only. This example should illustrate the amount of computations that can be saved by MOR. The mapping from $f(x, v, t)$ to $q(x, n, t)$ is implemented through a reduction algorithm, which also performs a remapping back to $\tilde{f}(x, v, t)$, such 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 $f(x, v, t) \in W$ is element of 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 [23]. 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 $q_n(\mu_i) \in V$ [5].



(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 the Sod shock tube. Next a 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 [23] and [5], 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. 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 the Sod shock tube chapter 2, decays with

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

where n denotes the number of RB or intrinsic variables. Note that hereafter we will solely use the term intrinsic variables[23]. The relevance of using non-linear reduction methods for MOR is often formulated in terms of a slow decaying Kolmogorov N-width. Altough BGK-model in the sod shock tube describes an advection problem with jump discontinuities chapter 2 implies that linear reduction methods should fail. But we will see in the following that this is only partly true for this study.

4.1 Offline Phase

4.1.1 Full order BGK-model

The FOM is the 1D BGK-model in the Sod shock tube for two levels of rarefaction, gratefully provided by Julian Köllermeier and the Departement of Mathematics at the RWTH Aachen. The Sod 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. Details about the BGK-model and Sod's shock can be found in chapter 2. One can be viewed as a slip flow [12] 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 [24]. The other is situated in the continuum flow regime with $Kn = 0.00001$ for which the Navier-Stokes equations can be utilized without hesitation. A detailed description can be found

Table 4.1: Problem setup for the BGK model in the Sod shock tube. The diaphragm is positioned at $x_d = 0.5025$. For $x < x_d$ the gas is present, for $x \geq x_d$ particles are absent.

Variable	Number of nodes i	Domain extension	Step size (uniform)
x	200	[0.0025,0.9975]	0.00499
v	40	[-10,10]	0.05128
t	25	[0,0.12]	0.005

in chapter 2. Hereafter the continuum flow will be referred to as **H**. Note that both **H** and **R** are three dimensional tensors containing $f(x, v, t)$.
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, Π_{FCNN}, is shown in eq. (4.4). Each row of Π_{FCNN} contains a sample shown to the FCNN. In turn we aquire $x_i t_i = 5000$ samples. The preprocessed matrix for the CNN, Π_{CNN}, is shown in eq. (4.5). We obtain $v_i = 40$ samples, each containing a matrix $\pi_{CNN}^{x_i \times t_i}$ holding the information about $f(x, t)$ for a fixed point in v. POD uses the Π_{FCNN} matrix or its transposition.

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

In the following a distinction between Π_{CNN} and Π_{FCNN} is omitted, when referring to the preprocessed matrices. However a distinction between the levels of rarefaction, namely **H** and **R**, will be utilized as Π_{h} for the former and Π_{r} for the latter. This designation is mainly used in appendix B and appendix A.

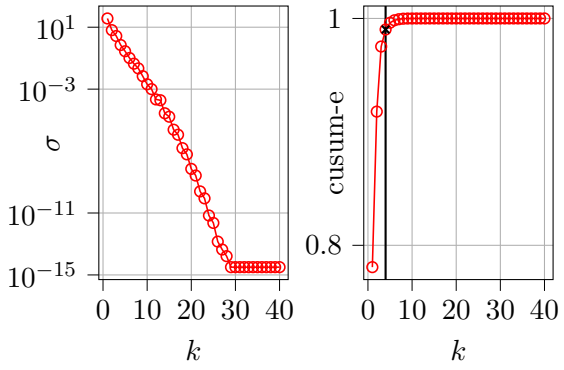
4.1.2 Contructing the mapping

With the FOM solution at hand it is possible to construct a mapping Q such that $Q(f) \approx \tilde{f}$. For Q, POD and two autoencoders, based on a fully connected neural network (FCNN) and a convolutional neural network (CNN), are employed, see chapter 3. The architecture, selection of hyperparameters and training for the FCNN and the CNN is discussed in appendix A and appendix B. Hence the availability of fully trained and tuned FCNN and CNN is given from now on. Since this thesis mainly focuses on methods from machine learning, POD serves as a comparative measure. To continue, the intrinsic variables obtained from POD, the FCNN and the CNN, will be referred as **h** and **r**. The former describes the intrinsic variables when reducing **H** and the latter when reducing **R**.
In order to contrast the number p of intrinsic variables in the autoencoders (sizes of **h** and **r**) we will utilize POD as a reference framework. Therefore the singular values σ, as well as, the cumulative energy (cusum-e)

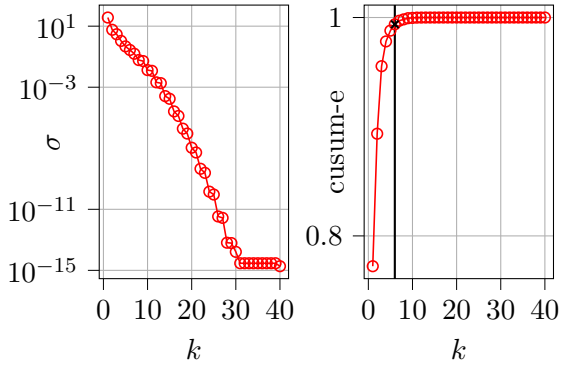
$$\text{cusum-e} = \frac{\text{cusum}}{\sum_i \sigma_i} \quad (4.6) \qquad \text{with} \qquad (\text{cusum})_i = (\text{cusum})_{i-1} + \sigma_i \quad (4.7)$$

over the singular values, are employed. A comparison for both levels of rarefaction is provided 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 $\sigma_6 = 0.275$. The rate at which the singular values drop is exponential in **H** and in **R**, with a difference of $L_2 = 0.025$ in the L_2-Error when comparing the gradients of the first $p = 10$ singular values. This should highlight the different characteristics when being reduced between both data sets.

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 solution. For both **H** and **R** the singular values and in turn the Kolmogorov N-width decay rapidly, which in turn leads to the assumption that advection and sharp shock fronts do not appear predominantly. Moreover the dissimilarly decay rate is a manifestation of the differing rarefaction levels. With a decreasing number of particles present, the lesser the probability of a single bulk macroscopic velocity emerges, see a full survey in chapter 2. Therefore **h** can always be chosen smaller than **r**, written as $\mathbf{h} < \mathbf{r}$, which is also evident with the FCNN and the CNN as discussed in the following.



(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** should be in total not more than three, as a slip-flow can be described in terms of three macroscopic quantities like density ρ, macroscopic velocity u and total energy E_{tot}[1][4]. Therefore $p = 3$ is employed for **h** and the autoencoders are arranged in that way. When considering **R** the picture becomes a little blurrier, as outlined before. More than one Maxwellian describe the microscopic velocities which increases p. To shed light into the size we want to choose for **r**, the size of the bottleneck layer of

the autoencoders, which is the same as p is varied in a next step.
To this end p is varied for POD, FCNN and CNN over $p \in \{1, 2, 4, 8, 16, 32\}$ with both **H** and **R**. 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 [6]. Shown in fig. 4.3 is the outcome of said experiments.

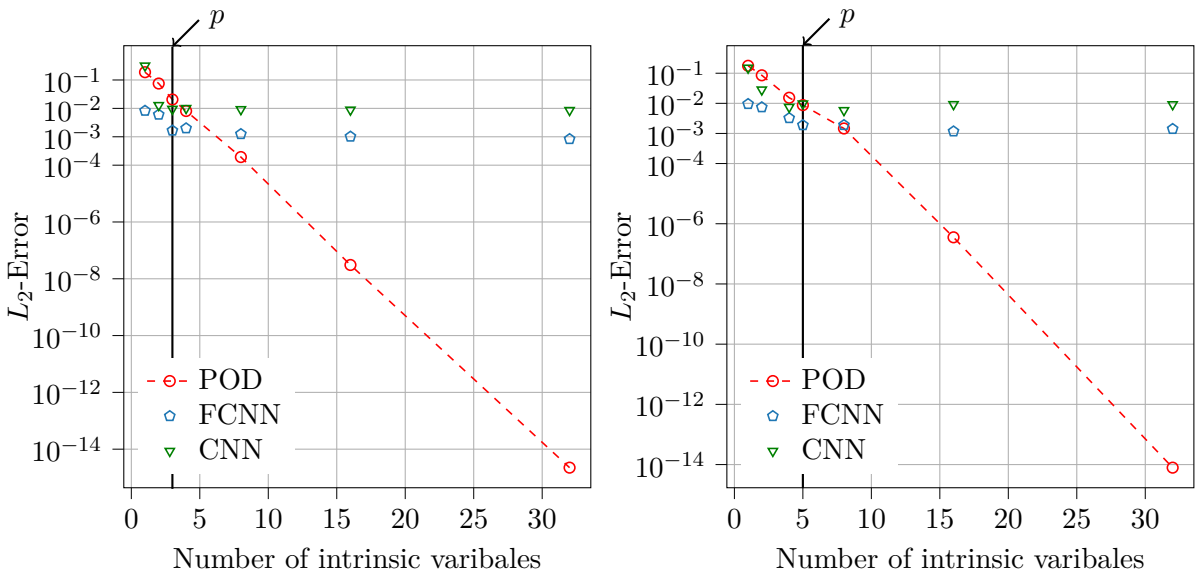


Figure 4.3: Variation of the number of intrinsic variables p over the L_2-Error for POD, FCNN and CNN. Results for **H** are displayed on the left and for **R** on the right.

The design for fig. 4.3 is taken from [5]. The loss of information when applying the 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 [3]. 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. Here a drop in the L_2-Error can be observed for both neural networks until $p = 3$. Stagnation is observed for the CNN afterwards. FCNN continues to improve slighly. However the choice for $p = 3$ can be confirmed.
Moving forward to establish the value of p for **R** we look at the right plot of fig. 4.3. A drop in the L_2-Error can be observed for the FCNN with increasing p until $p = 5$, which is highlighted with a black line. A continued widening of the bottleneck layer results in an slightly increased L_2-Error. Overfitting due to increased capacity of the model can be though of as the culprit here. Contrarily the CNN performs best with $p = 4$ and maintains a zig zag pace progression. The discrepancy between $p = 4$ and $p = 5$ in the CNN and FCNN is negligible. For one the FCNN performs overall better than it's convolutional counterpart, additionally shifts of missing nodes i.e. weights and biases in the bottleneck layer to the weights and biases of other layers makes this „method“ fairly inaccurate. Thus we decide for $p = 5$ in **R**.
A detailed survey of the reconstruction $\tilde{f}$ obtained from POD, the FCNN and the CNN using p is provided in chapter 5.

4.1.3 Online Phase

In the online phase new states of the FOM solution are calculated. Moreover new states need to be evaluated. With POD one usually exploits the intrinsic variables within a Galerkin framework as in [4] and [5] to produce new states, which won't discussed in this thesis. The same can be done with the intrinsic variables obtained from autoencoders as in [5]. In this contribution new states are produced employing simple interpolation in time t of $\mathbf{h}$ and $\mathbf{r}$. This approach can be thought of as the ability of the autoencoder to generalize, an idea that stems from deep learning. Therefore the generalization ability of the proposed autoencoder architectures will be analyzed in chapter 5.

5 Results

During the offline phase solutions obtained from the FOM, in this case $\mathbf{H}$ and $\mathbf{R}$, are reduced to their intrinsic dimension, as discussed in chapter 4. From the thereby obtained intrinsic variables which define a reduced basis namely $\mathbf{h}$ and $\mathbf{r}$, the solution can be reconstructed yielding a loss of information. Hence, before buiding a ROM, the reconstruction of $\mathbf{H}$ and $\mathbf{R}$ needs to be evaluated. If the reduction and subsequent reconstruction fails to sustain "important" information, then the reduction algorithm is not suited for building a ROM. What these "important" qualities for the BGK model in the test case are and how they can be measured, as well as methods to compare the FOM solution against it's reconstruction is discussed in this section. For the dimensionality reduction the proper orthogonal decomposition (POD), a fully connected neural network (FCNN) and a convolutional neural network (CNN) is used.

A first measure of how much information gets lost is obtained through one of our performance metrics during neural network training: the MSE over the validation set. Yet, this metric solely applies to neural networks. To be able to compare the POD to it's neural network counterparts another metric is in use: the error over the L_2-Norm here referred to as thr L_2-Error , already introduced in chapter 3. In table 5.1, the L_2-Error for all three algorithms on both input data is provided. The FCNN performs best out of the three both for $\mathbf{H}$ and $\mathbf{R}$ and even drops about one

Table 5.1: Comparison of the L_2-Error over the reconstructions obtained by POD, FCNN and CNN. The CNN was trained using the k-fold algorithm, therefore the mean μ over $k = 5$ folds, marked with superscript asterix, is given. Variance σ^2 for $\mathbf{H}$ is $\sigma^2_{\mathbf{H}} = 2.25 \times 10^{-5}$ and for $\mathbf{R}$ is $\sigma^2_{\mathbf{R}} = 3.61 \times 10^{-5}$. Parenthesised values are obtained from the best fold.

Algorithm	L_2-Error for $\mathbf{H}$	L_2-Error for $\mathbf{R}$
POD	0.0205	0.0087
FCNN	0.0017	0.0019
CNN	0.0142* (0.0095)	0.0178* (0.0097)

decade compared to POD and CNN for $\mathbf{H}$. Slightly better is the result for $\mathbf{H}$ than for $\mathbf{R}$ using both neural network designs. Yet the performance of FCNN and CNN doesn't seem to change a lot with changing $\mathbf{H}$ and $\mathbf{R}$. In contrast, the POD performs better for $\mathbf{R}$ than for $\mathbf{H}$. The reasons will be discussed hereafter. Note that the CNN is trained with the k-fold algorithm due to a lack of training samples as suggested in [6]. The k-fold algorithm is provided in appendix B. The mean μ over all folds gives an estimate of the models performance. Nonetheless the best performing fold is used in the subsequent analysis, yielding a better performance of the CNN.

The "important" qualities, mentioned in the beginning, become visible when looking at the worst reconstructions. Those can be determined with the L_2-Error over time for all three models as seen in fig. 5.1. For POD and the CNN the last timestep at $t = 0.12s$ in both cases **H** and **R** is the most

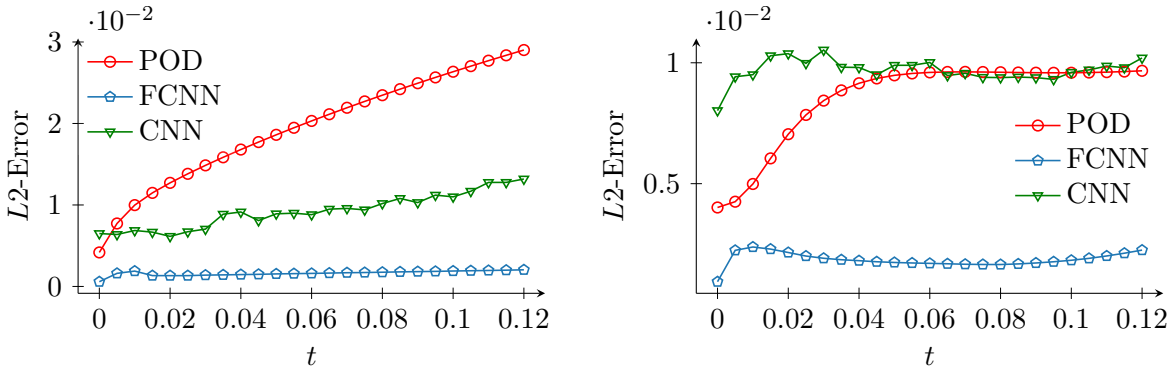


Figure 5.1: Relative Error over time for POD, FCNN and CNN. Results for **H** are displayed on the left, the results for **R** are displayed on the right.

rich in the $L2$-Error. While performing best, the FCNN in contrast has troubles in the beginning around $t = 0.01s$ for **H** and **R**. A detailed comparison of the models, reconstructions of $f(x, v)$ at $t = 0.12s$ and $x \in [75cm, 150cm]$ are given in fig. 5.2.

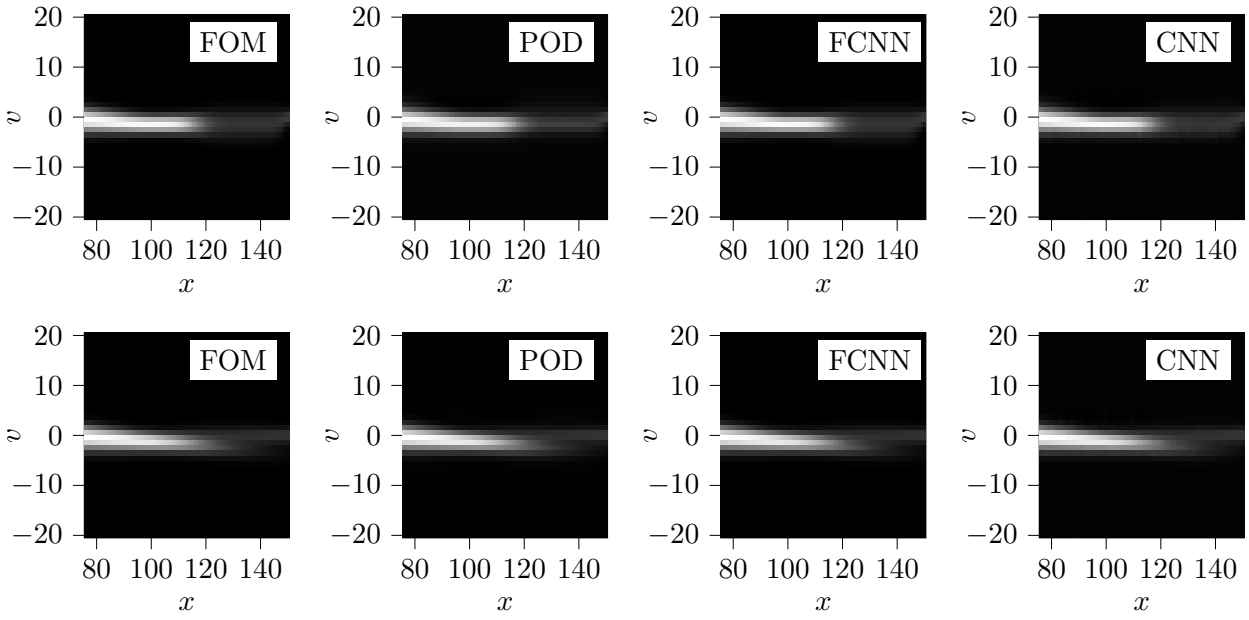


Figure 5.2: Comparison of FOM with three reconstructions obtained from POD, FCNN and CNN. The BGK-model in Sod's shock tube at time $t = 0.12s$ and $x \in [75\text{cm}, 150\text{cm}]$ of the tube, is most difficult to reconstruct by the aforementioned algorithms. Case **H** is displayed in the top row, **R** in the bottom row.

A presentation of the FOM in $f(x, v)$ was already introduced in chapter 2. For clarity, $f(x_0, v)$ is a probability distribution for a gas having a velocity v at point x_0 in space at one moment t_i in time. Top row in fig. 5.2 show reconstructions for **H** and the bottom row for **R** with FOM solution as paradigm. Starting with **H** one observers, that a restored $f(x, v)$ from $x = 120$, the point around which dilution initiates, gets defective for POD and the CNN. Here the probability distribution is thinner as the original with POD and smeared around the borders with the CNN. This in turn leads to errors in velocities gas particles can have once passing $x = 120$. In contrast the FCNN reproduces the FOM solution almost exactly.
Continuing with a row further down of fig. 5.2 and therefore **R**, in place of a pronounced separation in a dense and rare region, stands bifurcation of $f(x, v)$ into two probability distribution functions as outlined in chapter 2. POD and the FCNN reproduce the FOM solution without any visible drawback. On the other hand the CNN reproduces smeared corners as in **H**.

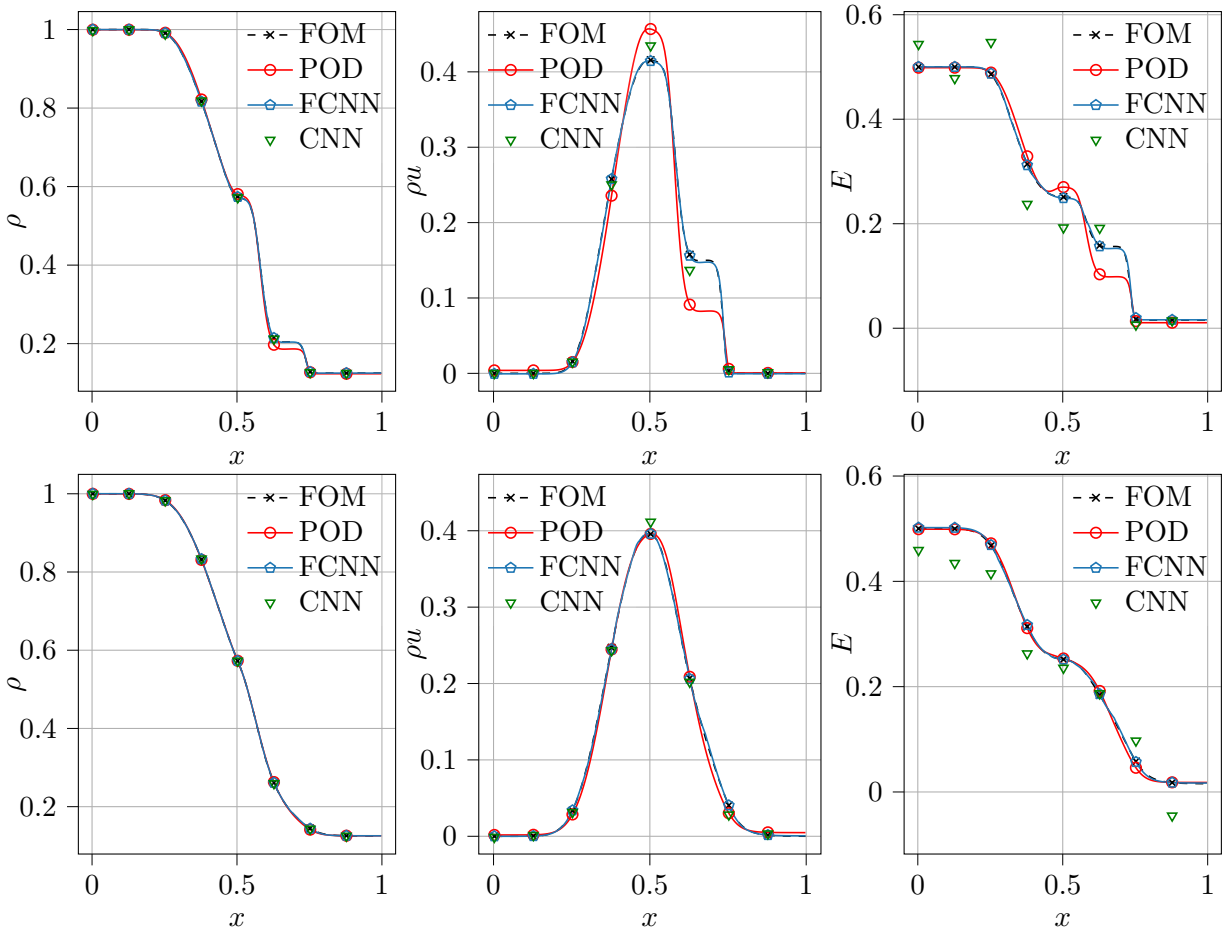


Figure 5.3: Matching of macroscopic quantities ρ, ρu and E reproduced by POD, FCNN, and CNN with FOM macroscopic quantities. Top row shows results for **H**, bottom row for **R**. CNN is displayed with marks only because of trembles in the signal.

Loss of information described above can unfold in severe mistakes in ρ, ρu and E, the macroscopic quantities, as displayed in fig. 5.3. In the following, 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 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, ρu and E, obtained by the FCNN is exact for both cases **H** and **R**. In particular the density ρ matches with results from the FOM exactly for the neural networks in both cases **H** and **R**. Second the CNN produces trembles in ρu and especially in E which is why it's shown with marks only. Results from the CNN are similar for **H** and **R**. The momentum ρu holds errors for the tail of the rarefaction wave as well as the contact discontinuity and the position and height of the shockwave. Third POD performs better on **R**, holding only small deviations in the contact discontinuity and position and height of the shockwave for the momentum ρu and the total energy E. The density ρ matches with the FOM solution exact. Distinct deviations from the FOM solution occur using POD on **H**. The density ρ holds errors in the height of the shockwave. The momentum ρu holds errors in the tail of the rarefaction wave, the contact discontinuity and the height of the shockwave. Hence in the total energy E errors occur in the rarefaction wave, especially the tail, the contact discontinuity and the height of the shockwave.

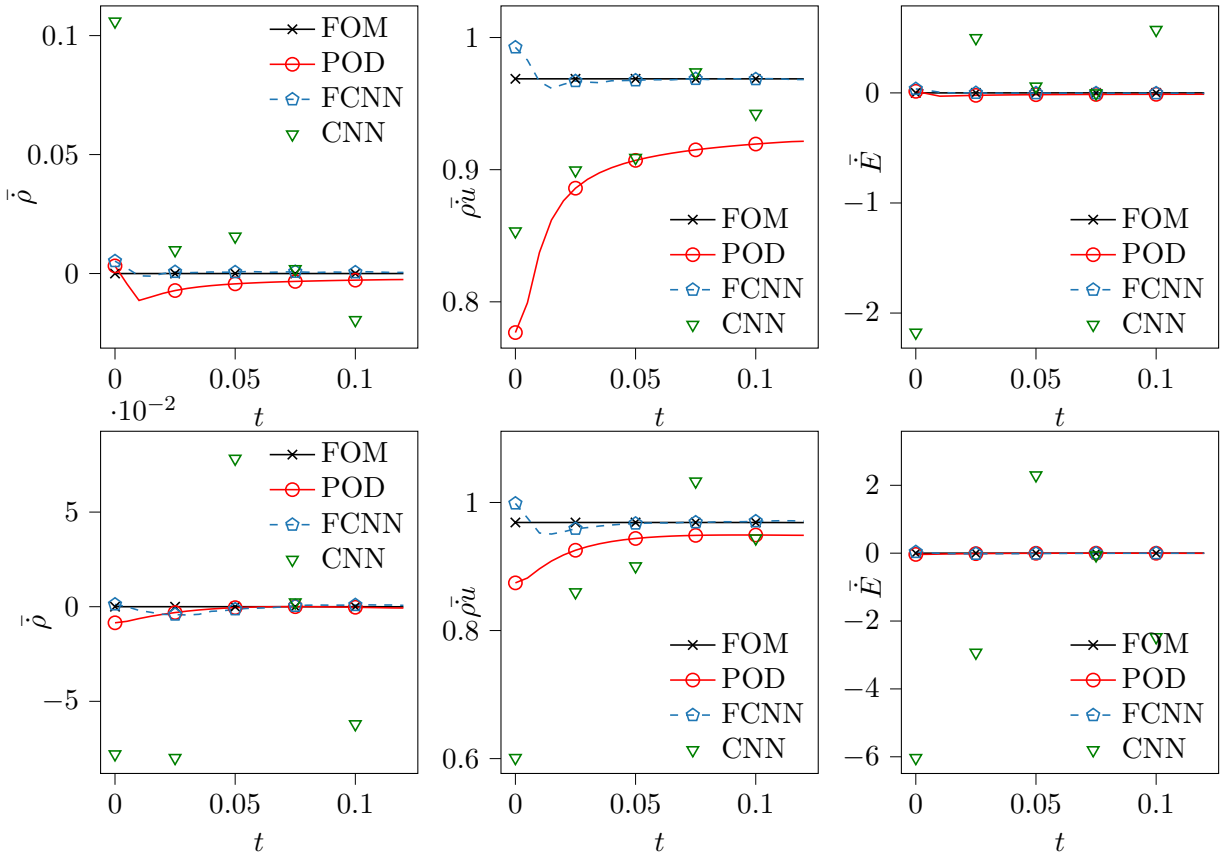


Figure 5.4: Comparison of the conservative properties of reconstructions obatined from POD, the FCNN and the CNN against the conservative properties of the FOM solution using the temporal mean.

Conservative properties of the FOM are discussed in chapter 2. Now we want to estimate if the conservation of mass, momentum and total energy could be sustained using POD, the FCNN, and the CNN. To do so, the temporal mean over the time derivative of the macroscopic quantities is employed. Figure 5.4 shows the conservation of mass, momentum and total energy over time 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 shows deviations from conservation with POD. Conservation of momentum meets the FOM solution after $t = 0.03s$ using the FCNN for both cases **H** and **R**. POD conserves momentum close to the FOM solution, but deviations are similarly present for **H** and **R**. Next 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.
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 **H** than with **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 **H** and **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.
We now reached the online phase where we want to be independent from the FOM solution. A first ROM relying on pure interpolation in the intrinsic variables is performed for **H**. The intrinsic variables of the FCNN for **H** are shown in ... .

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.

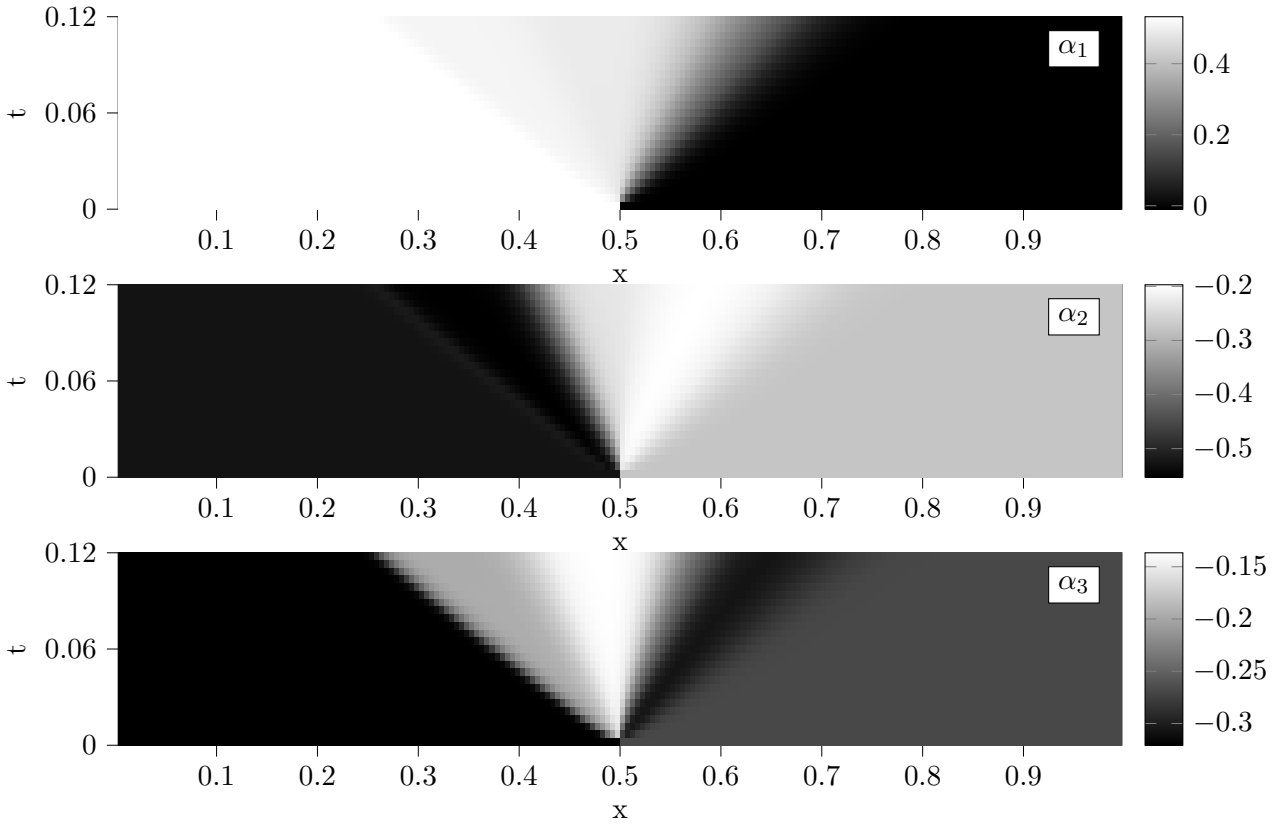


Figure 5.5: α_1, α_2 and α_3, the reduced basis **h** obtained from the FCNN.

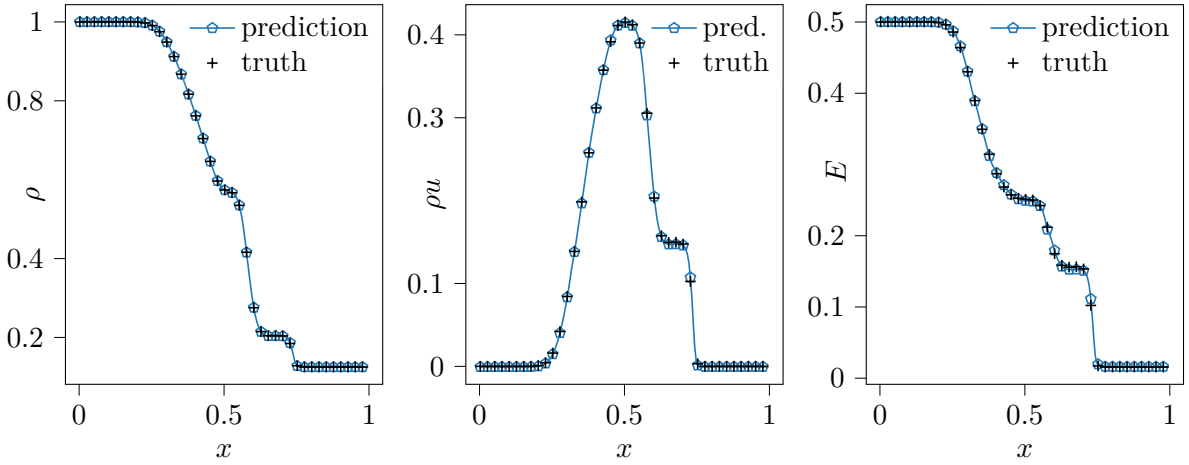


Figure 5.6: Interpoaltionin in time for **H** from 25 snapshots to 241 snapshots with cubic splines using the FCNN.

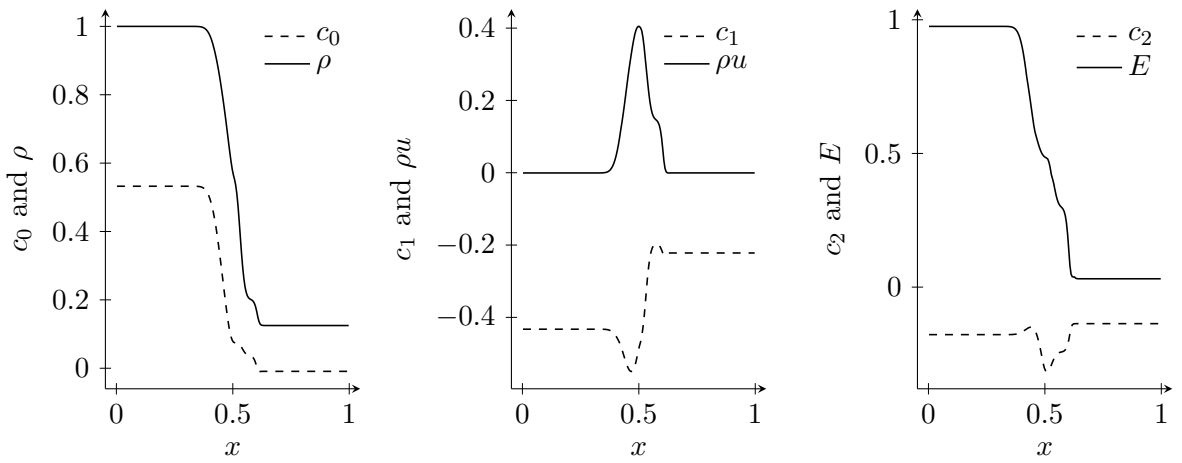


Figure 5.7: Code variables c_1, c_2 and c_3 (dashed lines - -) and macroscopic quantities ρ, E, ρu (full lines –) for $t = 0.05s$.

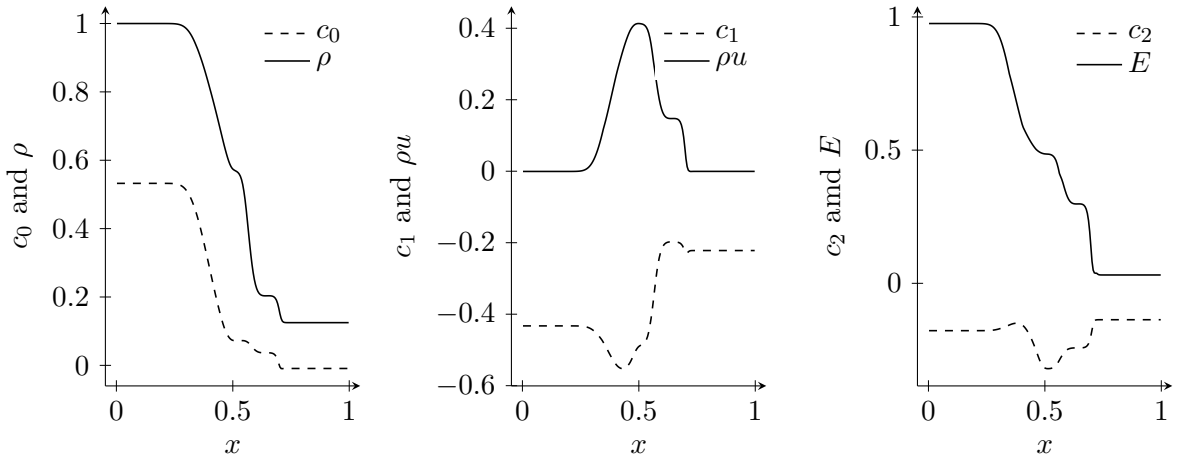


Figure 5.8: Code variables c_1, c_2 and c_3 (dashed lines - -) and macroscopic quantities ρ, E, ρu (full lines –) for $t = 0.099s$.

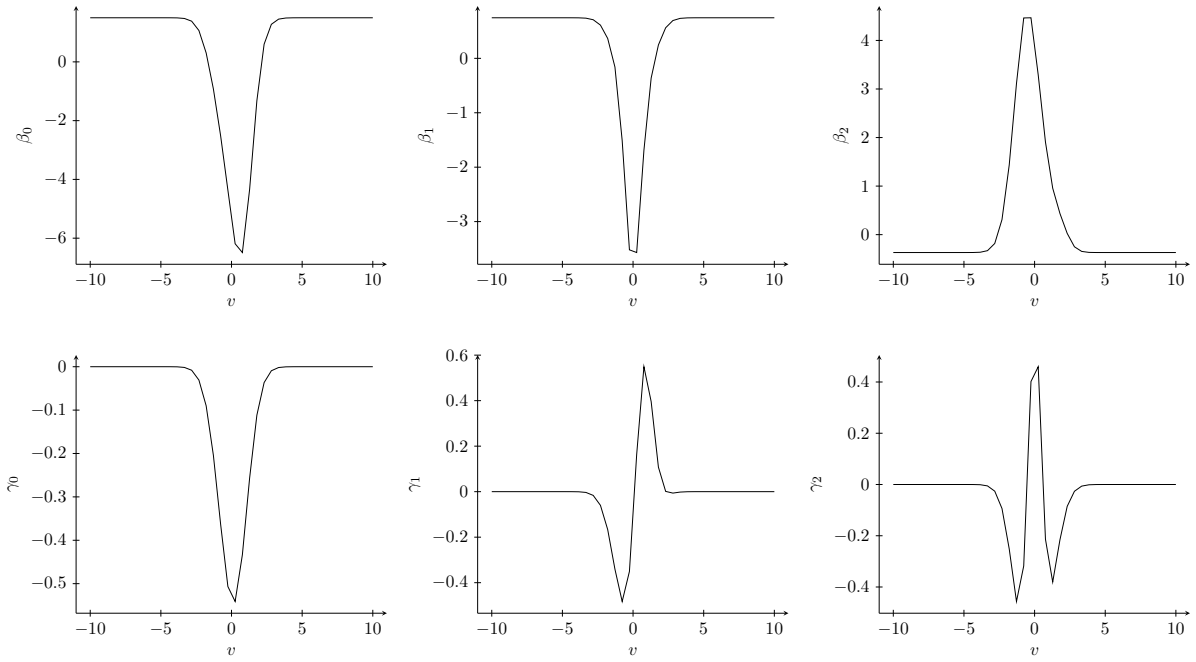


Figure 5.9: Comparison of the intrinsic variables generated by POD γ_1, γ_2 and γ_3 with the intrinsic variables of the convolutional autoencoder β_1, β_2 and β_3.

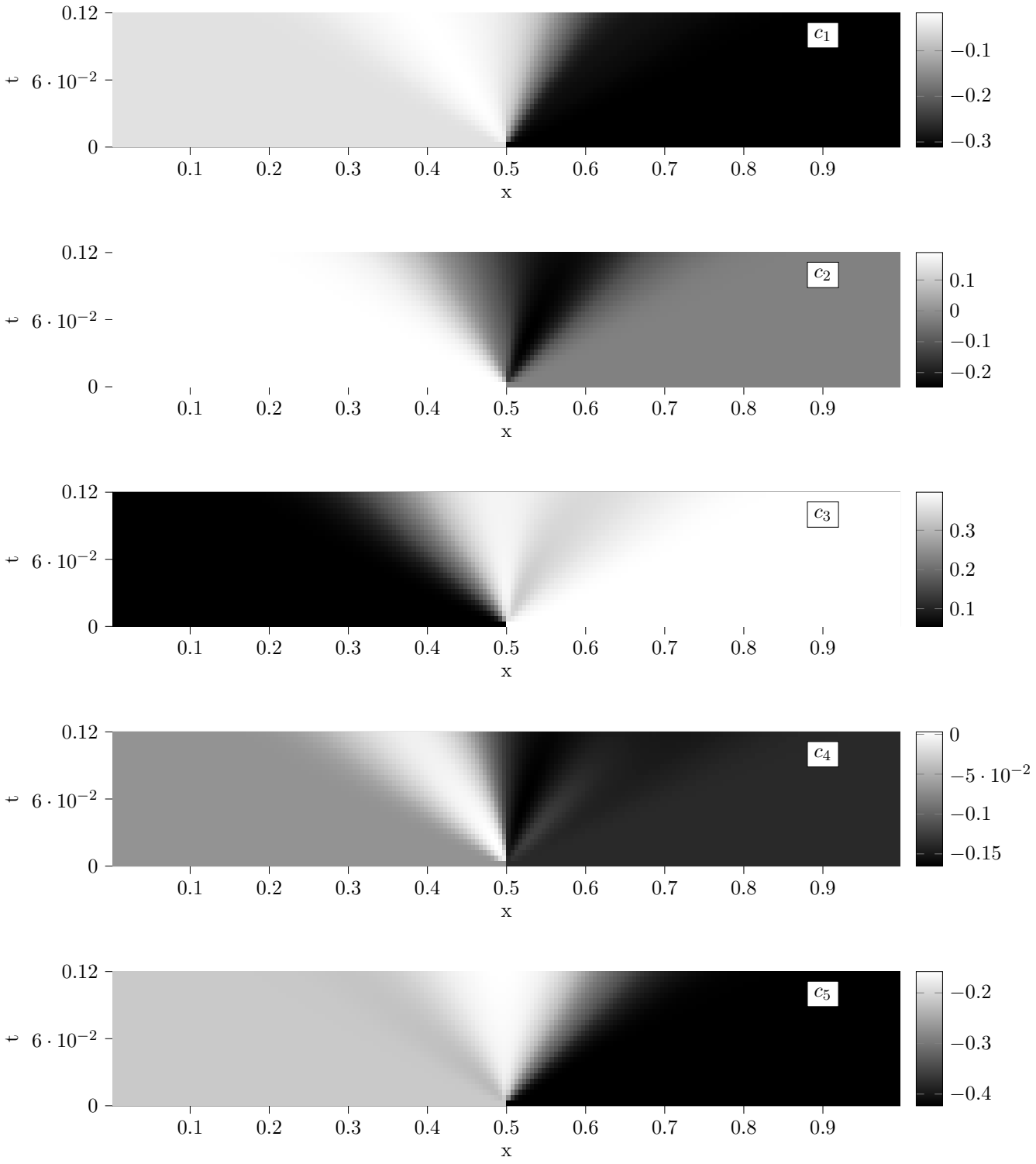


Figure 5.10: Intrinsic Variables of **R**

Appendix

A Hyperparameters for the Fully Connected Autoencoder

To start with a working model a guess is needed to be made about the initial design of the architecture. Here the hyperparameters already found are used as a starting point and shown in Table A.1.

To start with the number of layers or depth as described in **??** is determined, as they set a

Mini batch size	Intrinsic dimensions	Epochs	Learing rate	activation code/rest
16	3	2000	1e-4	Tanh/LeakyreLU

Table A.1: Initial hyperparameter selection

consequential part of the representational capacity of the model and therefore can initiate over- and underfitting at an early stage of the parameter search. **??** and **??** in **??** show the training and validation error for five designs shown in table A.2.

For the hydrodynamic regime a number of layers greater than four results in a slight overfitting

Number of layers	Reduction per layer
10	$40 \to 40 \to 20 \to 10 \to 5 \to 3$
8	$40 \to 40 \to 20 \to 10 \to 3$
6	$40 \to 40 \to 20 \to 3$
4	$40 \to 40 \to 3$
2	$40 \to 3$

Table A.2: Initial hyperparameter selection

at an early stage of training (at around 100 epochs). Below four layers an underfitting can be observed. Hence yielding the conclusion, that four layers result in the best perfoming net at this early stage. Overfitting occurs only after the 1000th epoch and is less than with the other three nets that show overfitting. Note, that contrary to expected results, the train error is lower than the test error. The random shuffling during preprocessing might be taken into account here. Nonetheless solely overfitting is defined by differences between train and test loss. In Addition four layers and lower show a stable training in relation to the rest.

The analysis of the number of layers for the rarefied regime is not showing overfitting as obvious as before. The network with two layers underfits in contrast to the other nets whereas nets with more than two layers reach a similar train - and test loss. Networks with more than 4 layers, again show an unstable training compared to the net with four layers. Train and test loss show a diverging behaviour after around the 100th epoch.

In conclusion the net with four layers performs best for both the hydrodynamic and the rarefied regime.

Training duration is raised form 2000 epochs to 5000, as training did now converge completely as seen in **??** and fig. A.2.
In the following the width of the hidden layer will be analysed. There are two available hidden layers for the autoencoder. But as the architecture of the decoder mirrors that of the encoder only one parameter needs to be varied. For both the hydrodynamic and the rarefied regime five experiments are conducted, varying the hidden units from ten to fifty. Results for Π_h and Π_r are shown in table A.3. For **H** and **R** forty hidden units performs best with an error around $1e^{-3}$ and

Hidden units	10	20	30	40	50
Error for Π_h	0.0054	0.0027	0.0036	0.0017	0.0032
Shrinkage factor for **H**	0.3	0.1	0.015	0.075	0.06
Error for Π_r	0.0078	0.0027	0.0022	0.0015	0.0025
Shrinkage factor for **R**	0.5	0.25	$0.01\overline{6}$	0.0125	0.01

Table A.3: L2-Error for different number of hidden units for Π_h and Π_r.

a shrinkage factor of 0.075 for **H** and 0.0125 for **R**. The combination of representational capacity set by the number of hidden units and shrinkage factor has an optimum here. When decreasing the number of hidden units, the representational capacity of the model also decreases. At the same time the shrinkage factor increases, therefore less information needs to be compressed in one step. The error increases with deceasing the number of hidden units from forty. This can be explained by the lower representational capacity. When increasing the number of hidden units to fifty the error also increases though, the representational capacity grows. Here the decreased shrinkage factor could be responsible for that.
Next the mini-batch size is analysed. Results are displayed in table A.4 for **H** and in table A.5 for **R**. Additionally appendix B.0.1 shows the training for both in put data **H** and **R**. For both input data experiments are first conducted with mini-batch sizes in the range of : $[2, 4, 8, 16, 32]$.
The results of the L2-Error show best performance between a mini-batch size of 8 and 16 for **H**. Likewise the results of the L2-Error for **R** show best perfomance between mini-batch sizes of 4 and 16. Therefore additional experiments are conducted. Three additional experiments for **H** with mini-batch sizes of: $[10, 12, 24]$. Two additional experiments for **R** with mini-batch sizes of: $[6, 10]$. The training reveals that smaller batch sizes lead to oscillating traing and test errors. This can be overcome by a smaller learning rate, but also leads to increased computational time as more epochs are needed to achieve a comparable training and test loss at the last epoch. A mini-batch size of 8 is bordering the zone where the oscillations of training and test loss subside. At the same time a mini-batch size of 8 to 10 indicate also a growth of the L2-Error. In conclusion a mini-batch size of 8 is chosen to continue to working with for **H** and **R**. The osccilations which make the training instable can be battled with a lower learning rate as soon as training starts to tremble.
Together with the mini-batch sizes, activations are examined. Hence experiments with different

Batch Size	2	4	8	10	12	14	16	32
Error for Π_h	0.0011	0.0011	0.0011	0.0029	0.0018	0.0018	0.0013	0.0030

Table A.4: L2-Error for different mini-batch sizes for Π_h.

activation functions are performed with the innitially used mini-batch size of 16. Besides the mini-

Batch Size	2	4	6	8	10	16	32
Error for Π_r	0.0014	0.0010	0.0012	0.0012	0.0014	0.0017	0.0017

Table A.5: L2-Error for different mini-batch sizes for Π_r.

batch size, epochs are as well taken from the inital selection of hyperparameters as 2000 epochs. Unlike the previous two examinations the selection of activation functions shows early in training if a acceptable level of perfomance can be achieved. This is a personal observation made throughout working on this thesis. First five activations are applied to all the four layers. Second a combination of activation functions is studied, where the intrinsic variables is activated with a different function than the remaining layers. Results can be observed in table A.6 and table A.7. Both input data

Activation function	ReLU	ELU	Tanh	SiLU	LeakyReLU
Error	0.0028	0.0019	0.0036	0.002	0.0039
Activation function	ELU/Tanh	LeakyReLU/Tanh	ELU/SiLU		
Error	0.0019	0.0017	0.0019		

Table A.6: L2-Error different activation functions and combinations for **H**.

Activation function	ReLU	ELU	Tanh	SiLU	LeakyReLU
Error	0.0048	0.0029	0.0037	0.0134	0.0.0034
Activation function	ELU/Tanh	LeakyReLU/Tanh	ELU/SiLU		
Error	0.0.0032	0.0030	0.0036		

Table A.7: L2-Error different activation functions and combinations for **R**.

lead to different results this time. For that reason **H** and **R** are studied separately. Starting with **H** and the values of the L2-Error, it can be observed that, while ELU and SiLU stand out with the lowest value of the L2-Error, when applying the same activation function to all layers, all three coupled activations convince in the combination case. Applying different activations to the layers increases the representational capacity of the model since the model can feed on a greater variety of functions as explained in the previous subsection. The need for a great representational capacity reduces for **H**. The reasons are the linearity of that case as described in [1]. Hence two activations (ELU and SiLU) from a similar family of functions perform well on **H** without the need of a second actiation. Still adding another activation render as good results concerning the L2-Error. When observing the train -and test loss in fig. A.5 one can notice that overfitting clusters the activations into two groups. While they all achieve a similar MSE-Loss for train -and test data only ELU and Tanh are not sowing overfitting. The combination of two activations functions yields an abundance of overfitting only for the combination of LeakyReLU with Tanh. In addition LeakyReLU with Tanh reaches together with ELU in combination with SiLU the lowest train -and test loss. The slight overfitting observed with the combination ELU and SiLU give rise to choose for the final model LeakyReLU with Tanh.
To continue with **R** the L2-Error is again observed in table A.7, ELU stands out for the single activation. ELU with Tanh and LeakyReLU with Tanh meet the lowest L2-error for the combination of activations. Next observing the training for the three activations in fig. A.6 one finds that all

three show little to no overfitting, while the combination of LeakyReLU and Tanh deviates from that slightly. From this point no clean decision can be made which activation or combination to take. Therefore another run is conducted for the three. Results are shown in table A.8. Only LeakyReLU with Tanh and ELU alone convince with a lower L2-loss. The train -and test loss does not give any hint which of the two remaining activations/ combination of activations to take concerning overfitting or a profound gradient for the train -and test loss. Hence a third run is performed taking the parameters already produced for the two models and training them for another 2000 epochs. This warm-starting of the models finally gives an answer on which one to take. The results in table A.8 show that LeakyReLU with Tanh, just as for **H**, perform well on **R** with the given model. In fig. A.7 the train and test loss during training are provided for teh second and third run.

Activation function	ELU	ELU/Tanh	LeakyReLU/TanH
Error 2nd run	0.0034	0.0029	0.0025
Error 3rd run		0.0024	0.0019

Table A.8: L2-Error different activation functions and combinations for **H**.

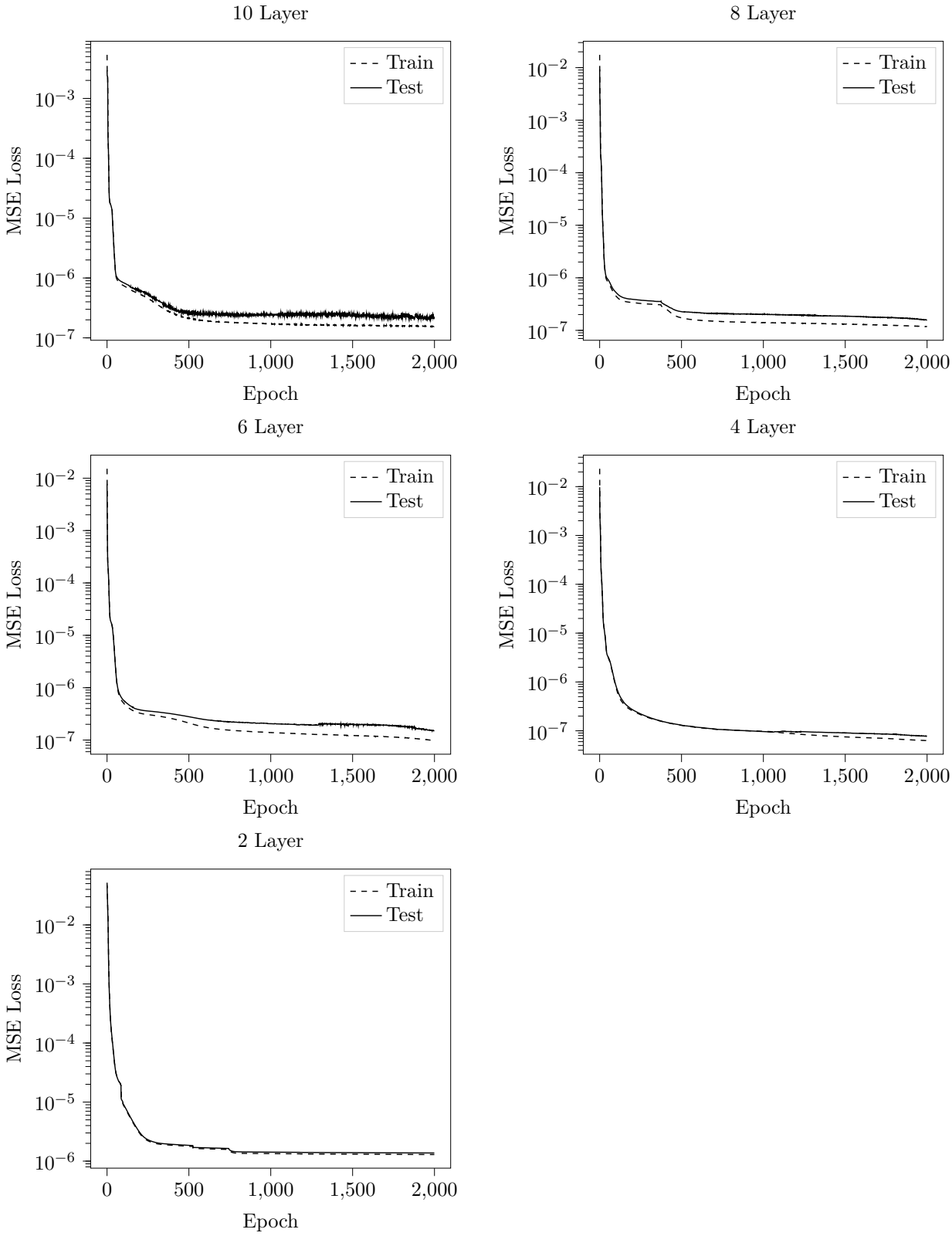


Figure A.1: Experiments with five different depth on **H**. Training and test loss are shown over 2000 epochs.

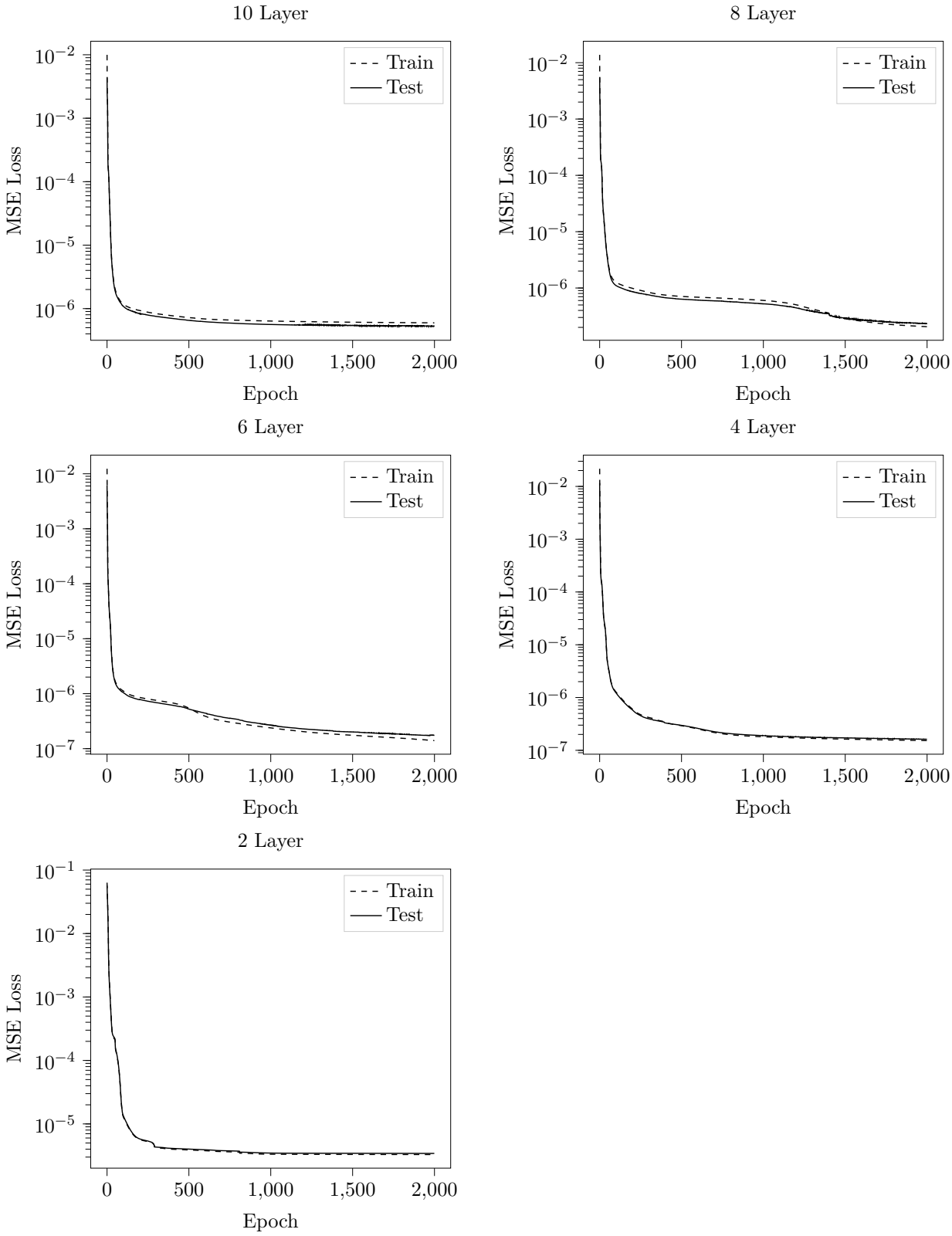


Figure A.2: Experiments with five different depth on **R**. Training and test loss are shown over 2000 epochs.

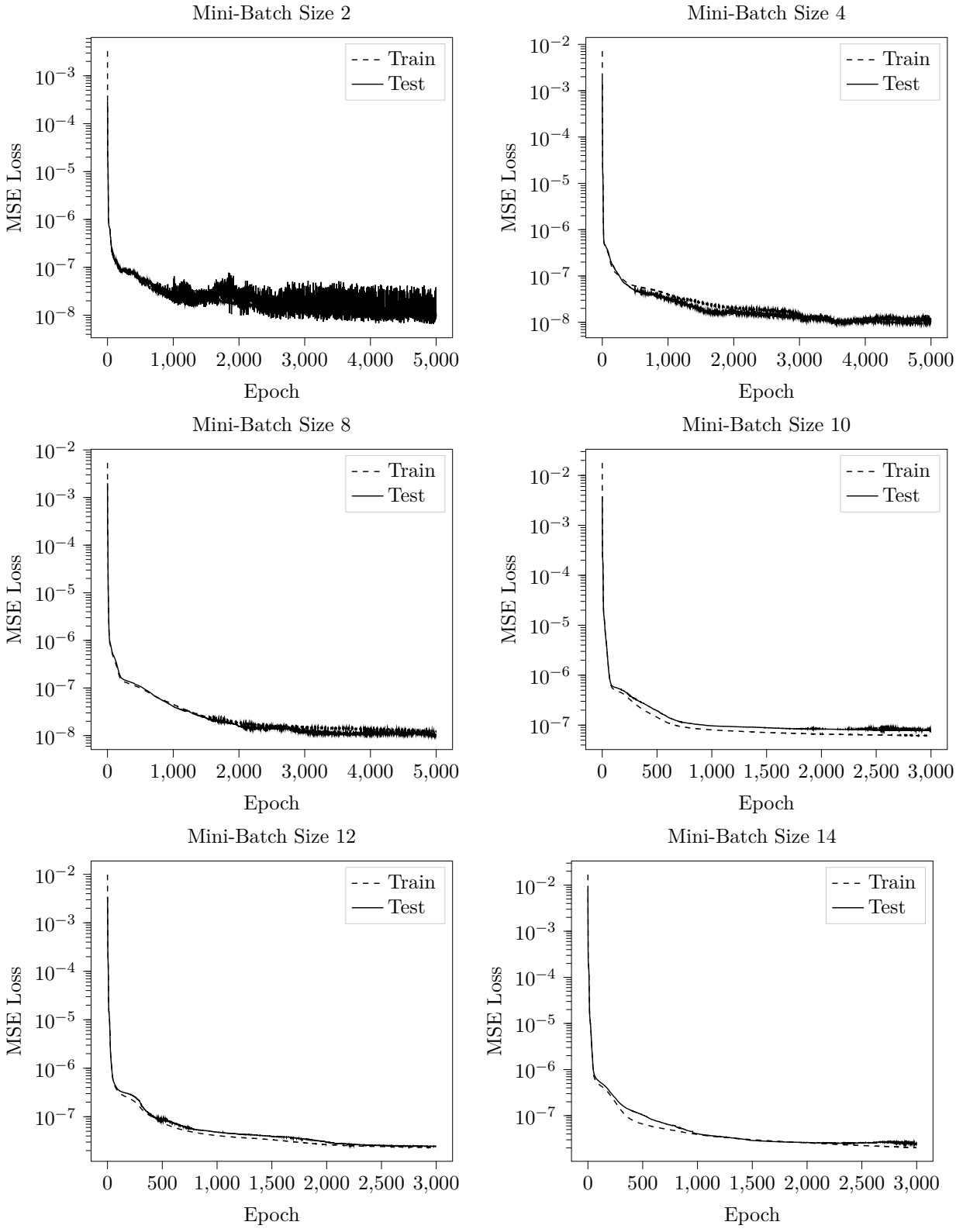


Figure A.3: Training with different mini-batch sizes for **H**. Train -and test loss is shown over 5000 epochs.

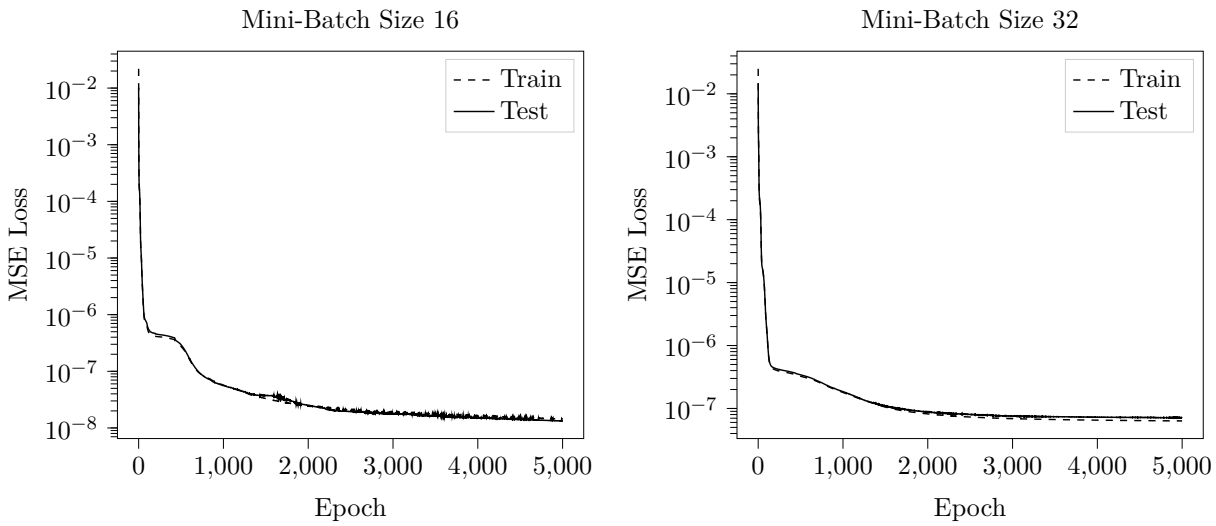


Figure A.3: Training with different mini-batch sizes for **H**. Train -and test loss is shown over 5000 epochs.

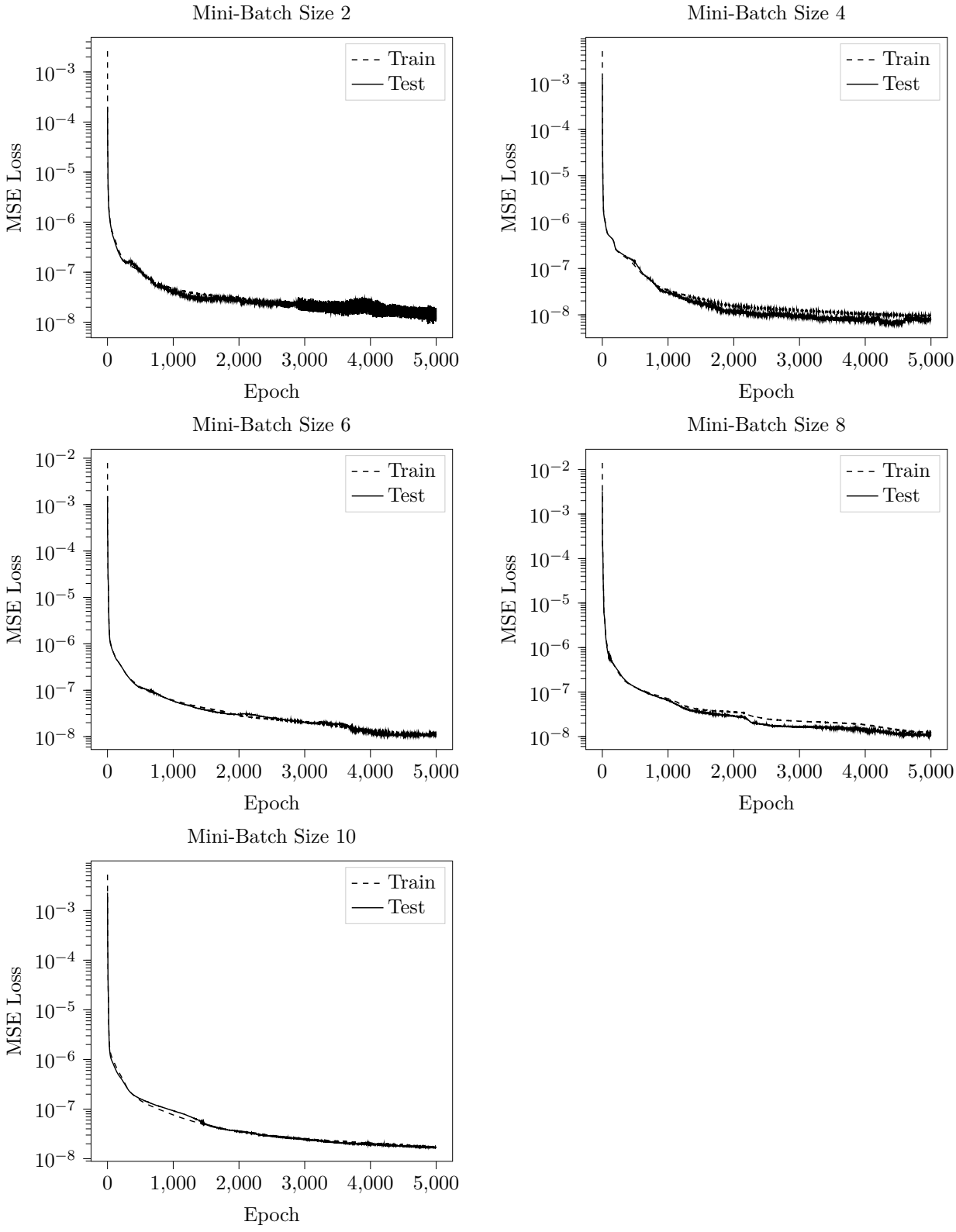


Figure A.4: Training with different mini-batch sizes for **R**. Train -and test loss is shown over 5000 epochs.

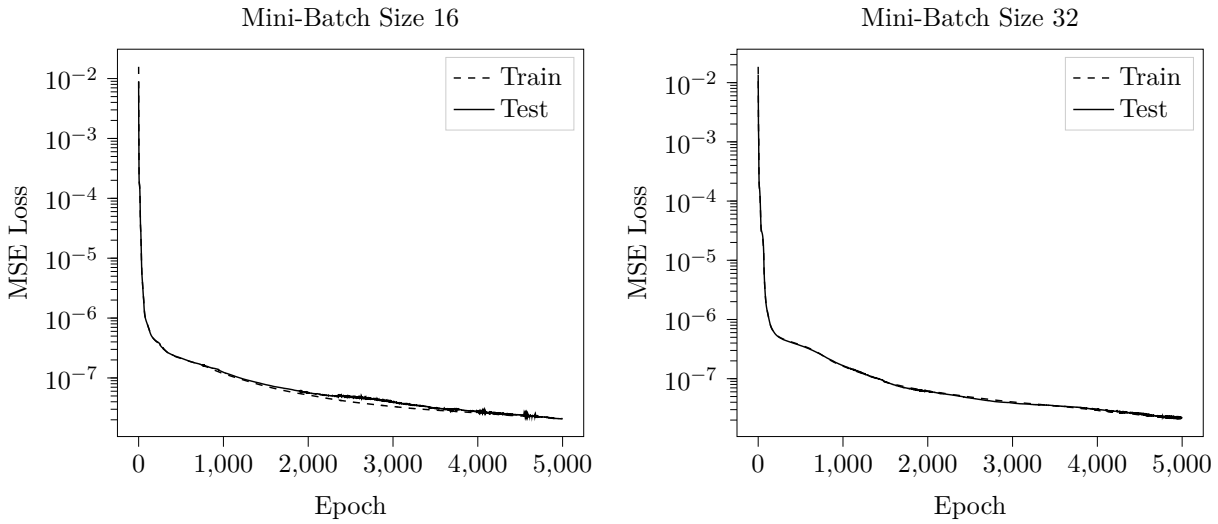


Figure A.4: Training with different mini-batch sizes for **R**. Train -and test loss is shown over 5000 epochs.

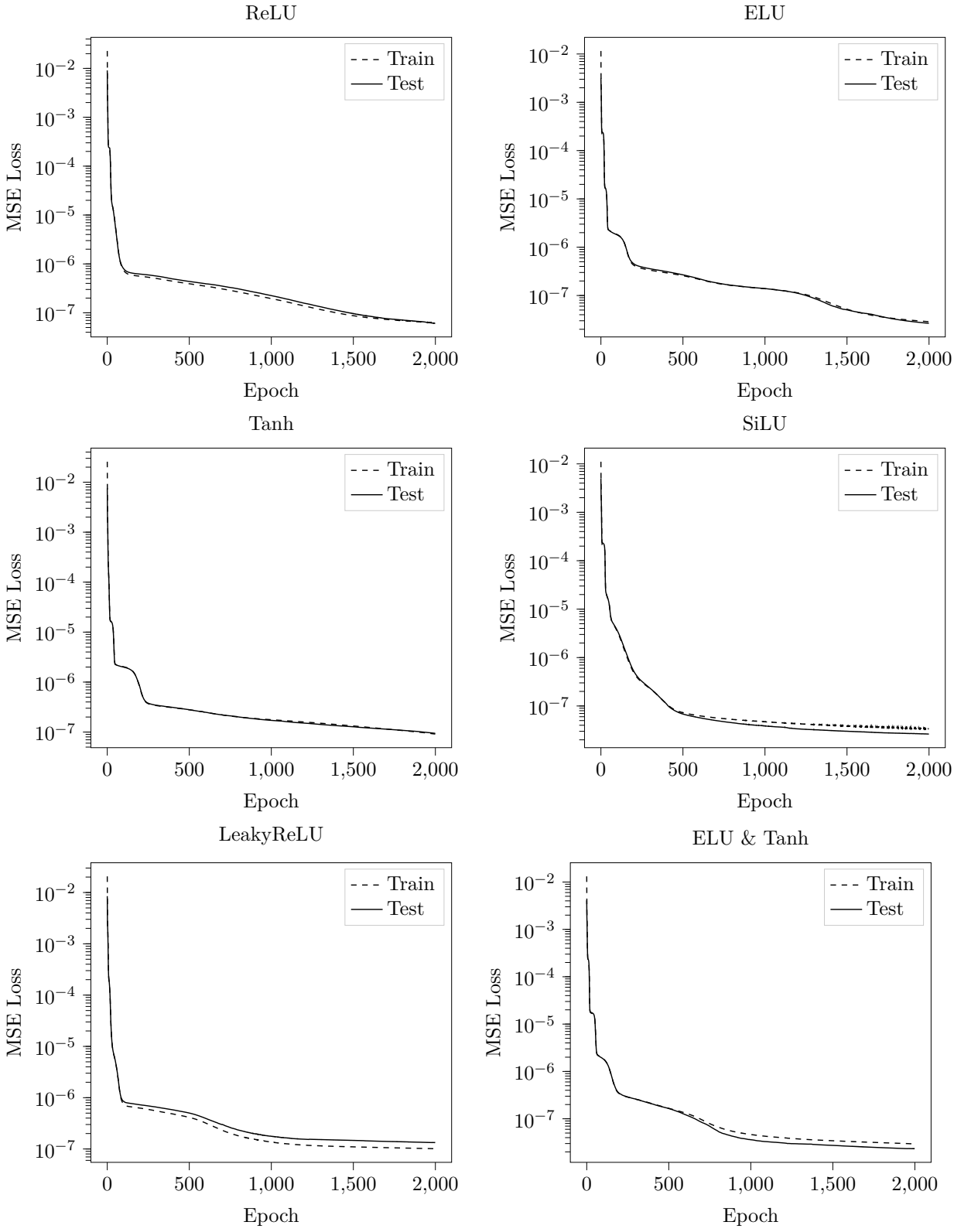


Figure A.5: balblabla

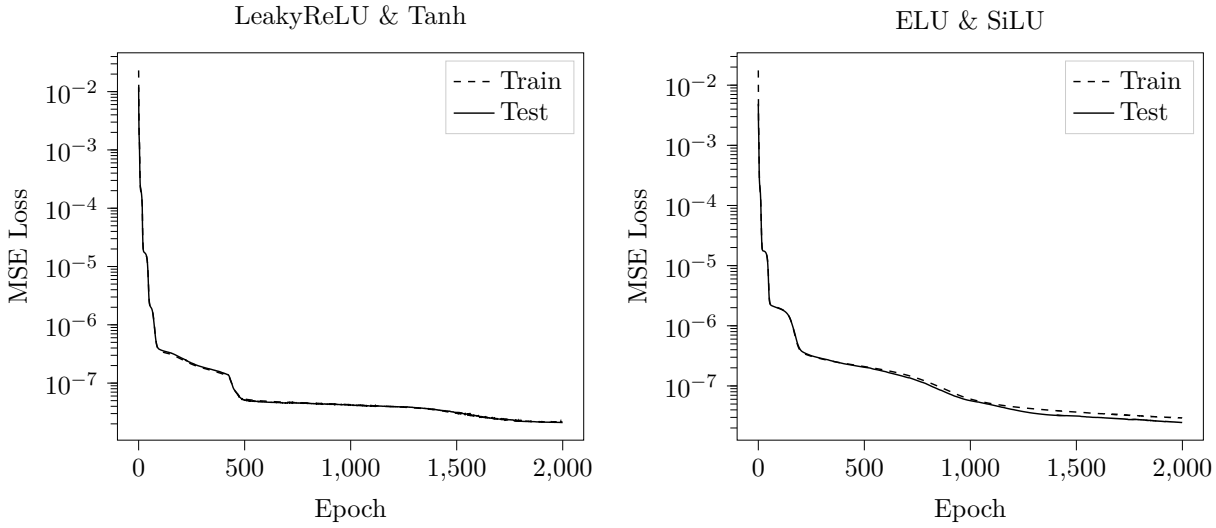

LeakyReLU & Tanh
ELU & SiLU
Train
Test
MSE Loss
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
0
500
1,000
1,500
2,000
Epoch

Figure A.5

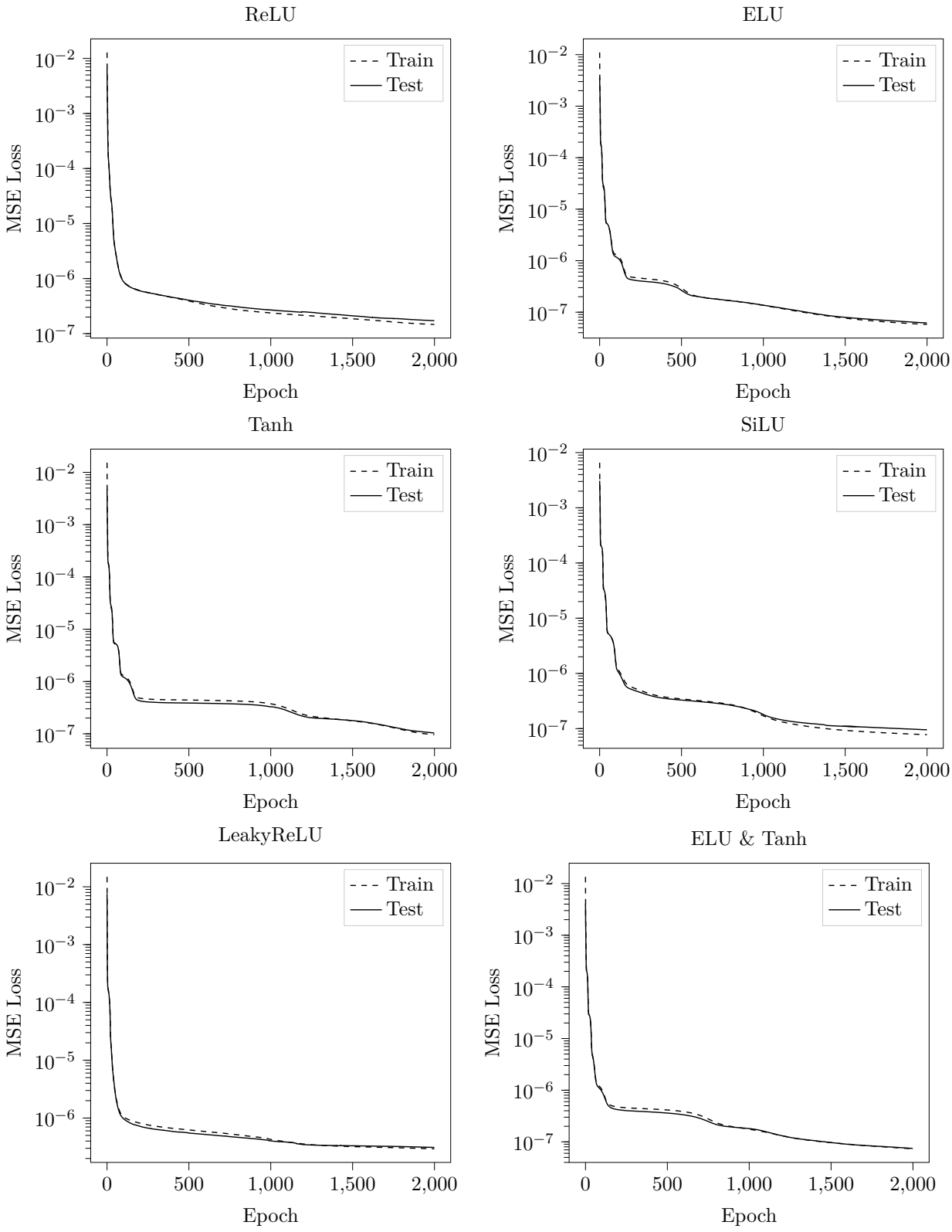


Figure A.6: blablabla

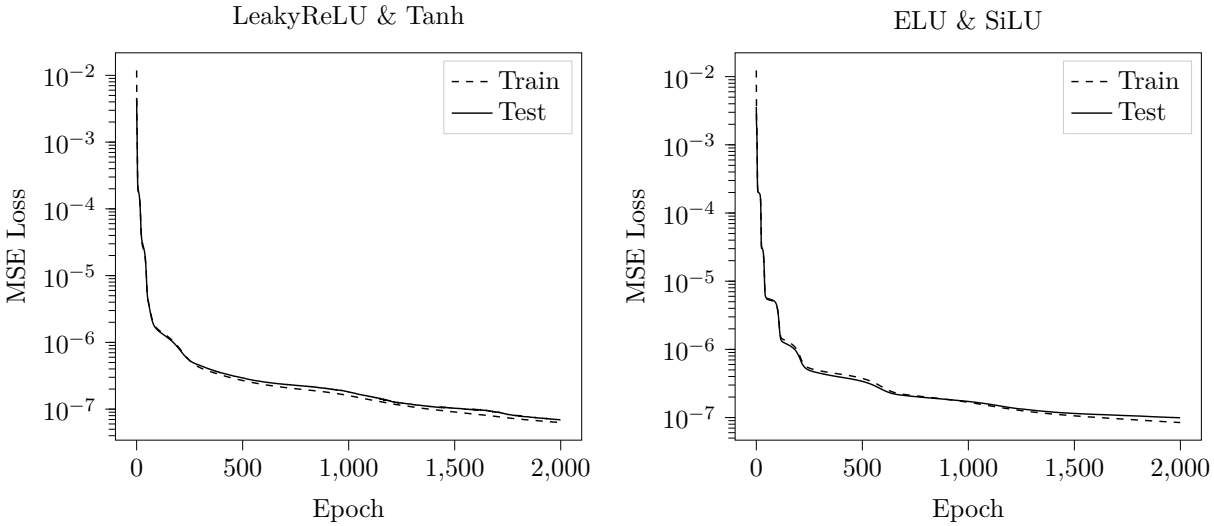


Figure A.6: blablabla

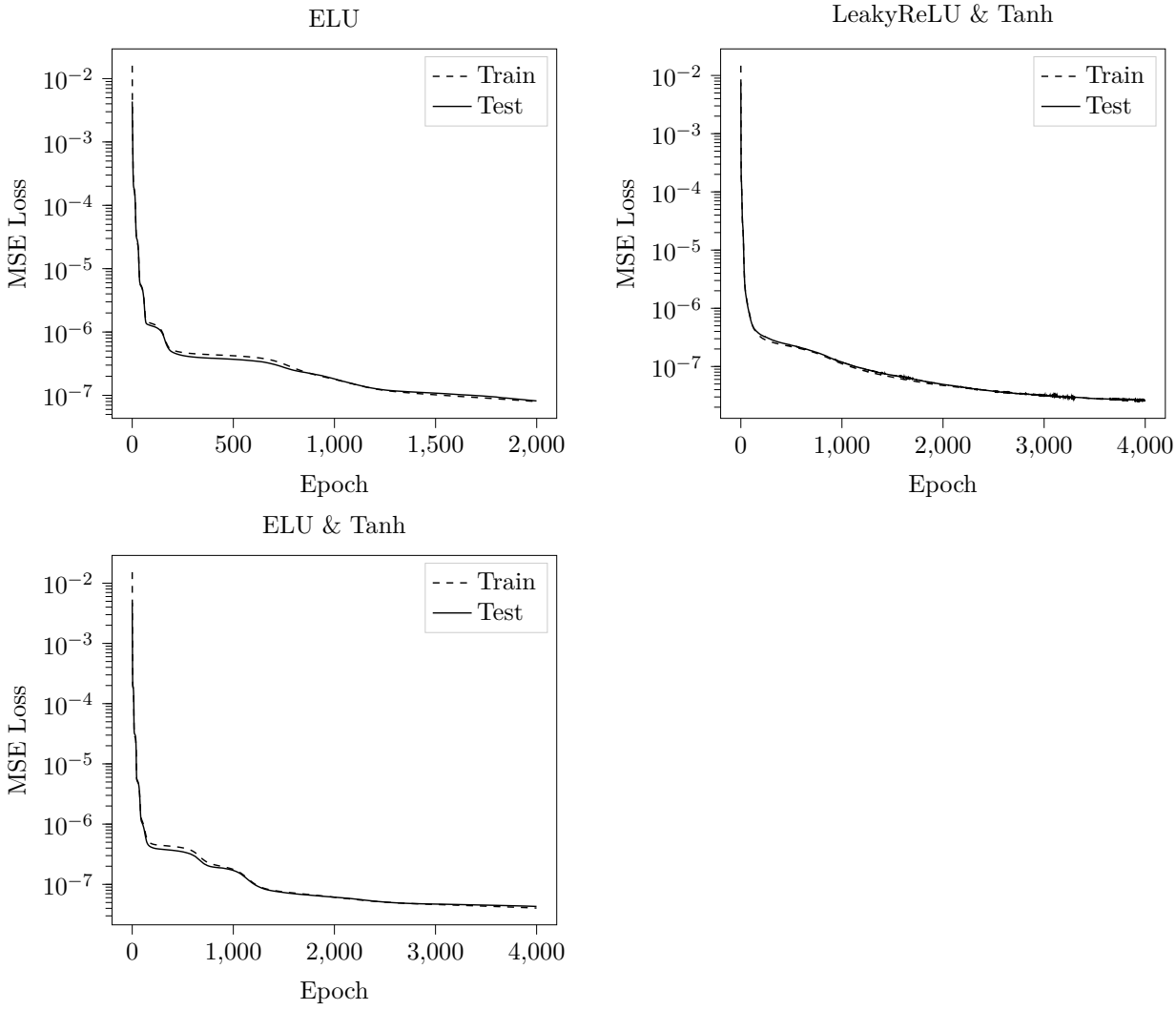
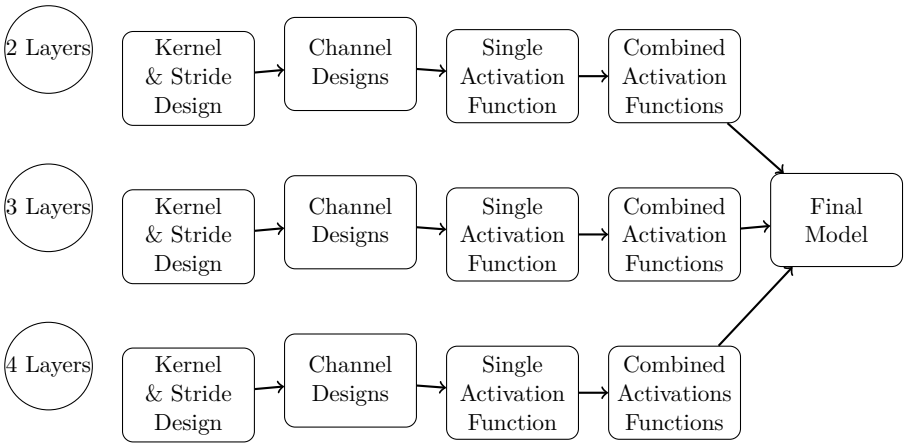


Figure A.7: blablabla

B Hyperparameters for the Convolutional Autoencoder



B.0.1 Appendix B

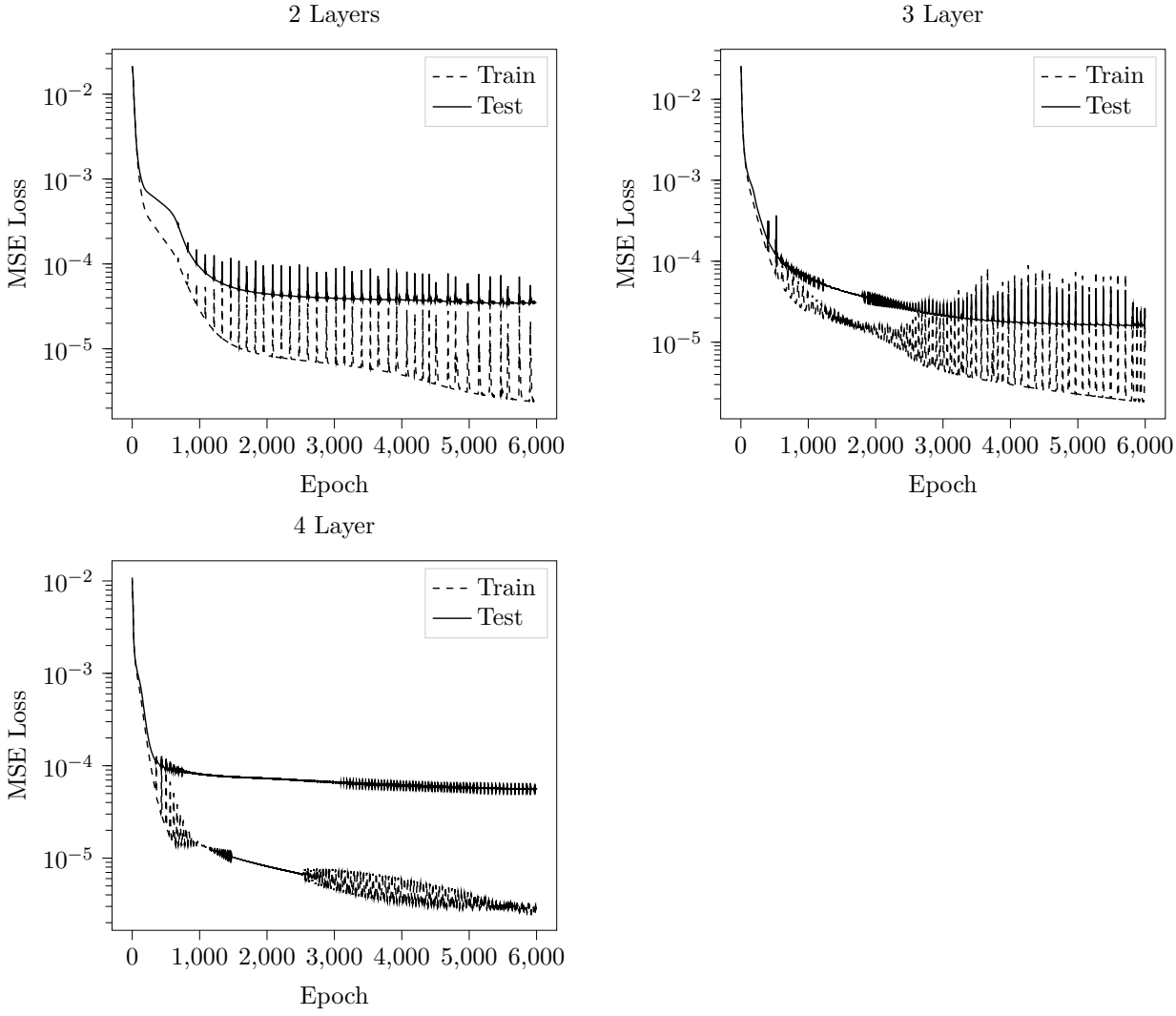


Figure B.1: Three convolutional networks with differing depth and with identical kernel evolution for **R**.

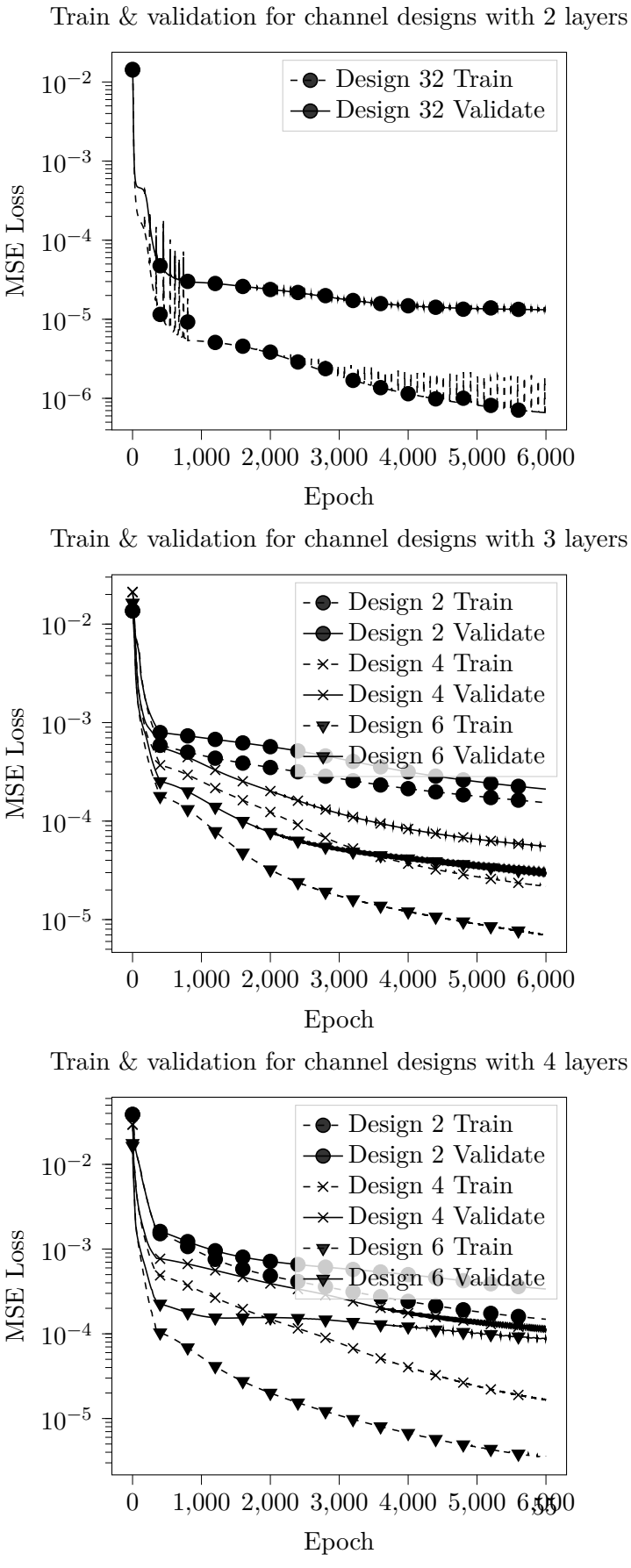


Figure B.2: Different Channel sizes for three convolutional networks with differing depth for **R**.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

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

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

[21] V. Singh, "basics of convolutional neural networks."

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

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

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

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