Convolutional autoencoder based reduced order modelling for physics problems
- master thesis report -

Hans van Malsen

January 2022

Master thesis report
Master thesis project 'Applied Physics'
Delft University of Technology

Supervisors:
Dr. Zoltán Perkó
Oscar Pastor Serrano

Abstract

Numerical solving a full order model can be computationally and time expensive. For real time control problems, it may be infeasible to solve full order models. Reduced order models can be used in order to reduce the time and computational cost while maintaining a high enough accuracy. In this thesis, it will be researched if a convolutional autoencoder based reduced order model is a feasible reduced order modelling method. Reduced order models will be constructed and applied for three different steady state neutron diffusion problems. Every autoencoder receives full order model solutions at its input. Convolutional layers are employed to process the high dimensional input to lower layers. The encoder will map the input data to the low dimensional latent space. The decoder will subsequently reconstruct the high dimensional input at its output from the low dimensional latent space. The latent space between the encoder and decoder forces the autoencoder to capture all necessary information in the few latent variables in such a way that the decoder can reconstruct the full order solution as good as possible. In order to find the optimal values for the model parameters, the autoencoder is trained on a set of full order solutions via gradient descent. After the training, the decoder can be used separately to map from the latent variables to the full order solutions. By joining the decoder with a regression model from the full order model parameters to the latent variables, one can find the full order solution without having to use a full order model solution method, like the finite element approach. In this thesis, a multivariate polynomial regression model is used for the regression from the full order model parameters to the latent variables. The convolutional autoencoder based reduced order model which incorporates residual blocks and parallel residual blocks in its structure, managed to outperform its proper orthogonal decomposition based counterpart by having an approx. 2.5 smaller mean squared error and a 1.4 times smaller mean absolute error. This shows that the proposed method is feasible in terms of prediction performance. Research should be done on the feasibility in terms of the computational costs and time costs. Additional recommendations are the extension of the proposed method to time dependent problems and the application to problems which are harder to capture with proper orthogonal decomposition based models.

Table of contents

1 Introduction **4**

2 Goal **4**

3 Related Work **5**

4 Theory **6**

4.1 Reduced order modelling . . . 6

4.1.1 Proper orthogonal decomposition . . . 6

4.2 Neural Networks . . . 7

4.2.1 Working principle . . . 7

4.2.2 Layers . . . 8

4.2.3 Optimization . . . 13

4.3 Heuristics of neural networks . . . 17

4.3.1 Over-fitting and under-fitting . . . 17

4.3.2 Weight initialization and batch sizes . . . 18

4.3.3 Activation functions . . . 19

4.3.4 Learning Rate . . . 20

4.3.5 Regularization . . . 21

4.4 Network architectures . . . 21

4.4.1 Residual blocks . . . 21

4.4.2 Inception blocks . . . 22

4.4.3 Autoencoders . . . 23

4.5 Neutron diffusion problems . . . 24

5 Methodology **27**

5.1 Method . . . 27

5.2 Application examples . . . 30

5.2.1 One group neutron diffusion with cross-section perturbations . . . 30

5.2.2 Multigroup problem with cross-section perturbations . . . 31

5.2.3 One group problem with geometric perturbations . . . 32

5.3 Proposed network structure: parallel residual block . . . 33

5.3.1 Used models . . . 34

6 Results **35**

6.1 Diffusion problems . . . 35

6.1.1 One group problem with cross-section perturbations . . . 35

6.1.2 Multigroup problem with cross-section perturbations . . . 38

6.1.3 One group problem with geometric perturbations . . . 41

6.2 Other results . . . 45

6.2.1 Parallel resBlocks, residual blocks and no skip-connections . . . 45

6.2.2 Regularization . . . 46

6.2.3 Data sparsity . . . 47

6.2.4 Batch normalization, layer normalization and no normalization . . . 48

6.2.5 Upsampling layers . . . 49

6.2.6 Custom activation functions . . . 49

7 Discussion **51**

8 Conclusion **52**

References **53**

9 Appendix A **55**
9.1 Extra figure one group problem with cross-section perturbations 55
9.2 Extra figures multigroup problem with cross-section perturbations 56

1 Introduction

Physics problems can often be described by a set of partial or ordinary differential equations. Some of those differential equations can be solved analytically, but most have to be solved by numerical methods. An accurate and reliable method of solving is done with a finite element method. The solution of a finite element method is called a full order model (FOM) solution. Many disciplines utilise the finite element method including, fluid dynamics, structural analysis, thermodynamics, electrodynamics and reactor physics.

Unfortunately, solving a FOM can be expensive in terms of time and computational power. Reduced order models (ROMs) are used to lower those costs while maintaining a certain accuracy. A common ROM method is the proper orthogonal decomposition (POD). This linear operation requires FOM solutions, which it will decompose in POD modes. The most dominant modes will be saved. All FOM solutions will be approximated by a linear combination of those saved modes. The POD coefficients plus the modes are enough to accurately describe a FOM solution. The coefficients can be used for further calculations, e.g., the time evolution of a FOM solution can be approached by the time solution for the POD coefficients. Since the number of used POD coefficients is usually much lower than the number of elements of a finite element FOM, one could speak of a ROM.

An alternative way of reducing a FOM to a small number of variables / coefficients is done with autoencoders (AEs). AEs are a type neural network and are already used for data compression. With the right settings, the reduction will be a nonlinear operation, allowing the AE to map to and from nonlinear variables which are called the latent variables. In this thesis, research will be done on the use of convolutional autoencoders (CAEs), which are AEs that make use of convolutional layers. Is a CAE based ROM able to outperform a POD based ROM in terms of accuracy?

This question and the goal of the thesis will be further specified in Section 2. Subsequently, some related work will be mentioned in Section 3. The necessary theory is given in Section 4. This is followed by methodology in Section 5 and the results in Section 6. The results will be discussed in Section 7 and the thesis will be concluded in Section 8.

2 Goal

The goal of the thesis is described as follows: Construct a CAE based ROM for physics problems. This can be further specified in two parts, where firstly a CAE needs to be constructed in such a way that its decoder can reconstruct the input for a limited latent space dimension, and where secondly, a regression model needs to be constructed to directly find the latent variables from the FOM parameters. The main focus will be on the construction of the CAE and less on the regression model from the FOM parameters to the latent variables. Time dependent problems are not covered.

3 Related Work

Autoencoders have been used in similar applications before. E.g., [1] has created a shallow, sparsly connected AE to reduce the dimensions of a 1D and 2D Burger's problem solutions to a nonlinear subspace, i.e., the latent space of the AE. The latent variables are used as coordinates for the time integration performed with the Galerkin method and the Least-Squares Petrov Galerkin method. They numerically show that for the two applied Burger's problems the nonlinear subspace representation (found with the encoder from the AE) outperforms its linear subspace counterpart (found with a POD variant). In a similar fashion, [2] used a fully connected AE to reduce the order of a compressible Navier-Stokes problem, namely 2D flow around a cylinder, as well as a shallow water problem to model the tides in the bay of San Diego. They predicted the time evolution of the latent variables of both problems by a NODE, which is a neural network used for solving ODEs. The ROMs of both applied problems were compared with o.a. POD based ROMs, which used a POD instead of a encoder to map to a low dimensional subspace. The time evolution was still done via a NODE. They found that the AE reduced subspace would yield better results for both problems. Another way of reducing the dimensions of a problem was proposed by [3], who used a Jacobian / derivative information to reduce the dimension of a problem and used a POD projection for the outputs. In between those two models is a neural network which aims to further reduce the dimensionality. But by already applying dimensionality reduction before the input of the neural network, only layers for small dimensionality need to be trained. A sparsly connected dense AE was used by [4] with the goal of finding a coordinate transformation such that the new coordinates would be a linear representation of nonlinear dynamics. The new coordinates are the latent variables of the AE and the coordinate transformation is done by the encoder. This is not a ROM, but shows that the encoder can be used to find lower dimensional representations. A CAE was used by [5] to reduce a 1D Burger's problem and a 2D inviscid shallow water problem into small latent spaces. For both problems, a Long Short-Term Memory (type of recurrent neural network, has backward connections) was constructed and trained to predict the time evolution of the latent variables. The results were compared to a POD combined with Galerkin projection. Another CAE based ROM was proposed by [6] where the high dimensional input was reduced to the low dimensional latent space via a convolutional encoder as well, but the time evolution of the latent variables was determined with Galerkin projection and Least-Squares Petrov Galerkin projection. They numerically showed that both ROMs outperformed its POD based counterparts on a 1D Burger's problem and a chemical reacting flow problem (model of H_2-air flame).

4 Theory

In this section the necessary theory will be presented. It starts with Subsection 4.1 about reduced order modelling, followed by large Subsection 4.2 about neural networks, their heuristics (Subsection 4.3) and some architectures of neural networks (Subsection 4.4). The Theory will be concluded with Subsection 4.5 about a specific physics problem, namely neutron diffusion.

4.1 Reduced order modelling

Almost all disciplines in science and engineering require some (physical) problem to be solved. Those problems often come in the form of a partial differential equation. Some partial differential equations can be solved analytically, but most have to be solved by numerical methods. An accurate and reliable method of solving is done with a finite elements method. This solution method yields the FOM solution. Many disciplines utilise this method, including fluid dynamics, structural analysis, thermodynamics, electrodynamics and reactor physics.
Solving a FOM is time and computationally expensive. ROM can be used to drastically downscale the time and computational burden while not reducing the accuracy too much. In cases where multiple FOM evaluations are simply infeasible, ROM can enable model evaluations. This makes ROM a powerful tool and it can be applied in many different fields, ranging from real time control systems to design optimisation and data assimilation. For example, in design optimisation, with reduced computation time, more model realisations can be calculated within a set time frame, yielding better optima in a given time.

4.1.1 Proper orthogonal decomposition

A common method of reducing the dimensionality of the FOM is by performing a POD. This method adds snapshots of FOM solution data in a matrix $\mathbf{S}$. Every FOM solution is flattened to be of dimension $\mathcal{R}^{m\times 1}$ and added to a column of the snapshot matrix. This means that if there are β different FOM solutions, the snapshot matrix will have the dimension of $\mathbf{S} \in \mathcal{R}^{m\times\beta}$. Subtracting the mean of every row yields matrix $\tilde{\mathbf{S}}$. This matrix will be decomposed by singular value decomposition as:

$$\tilde{\mathbf{S}} = \mathbf{U}\mathbf{\Sigma}\mathbf{B}^T, \tag{1}$$

where $\mathbf{U} \in \mathcal{R}^{m\times m}$ and $\mathbf{B} \in \mathcal{R}^{\beta\times\beta}$ consists of the eigenvectors of $\tilde{\mathbf{S}}\tilde{\mathbf{S}}^T$ and $\tilde{\mathbf{S}}^T\tilde{\mathbf{S}}$ respectively. $\mathbf{\Sigma} \in \mathcal{R}^{m\times\beta}$ is zero everywhere, except at the diagonal where the singular values σ_i are represented. According to [7], the basis functions $\boldsymbol{\phi}_i$ can be constructed as:

$$\boldsymbol{\phi}_i = \tilde{\mathbf{S}}\mathbf{B}_i/\sqrt{\sigma_i}, \tag{2}$$

where $\mathbf{B}_i \in \mathcal{R}^{\beta\times 1}$ is the i-th column of matrix $\mathbf{B}$. The basis vectors span the solution space and a linear combination of all of them will perfectly reconstruct any FOM solution of the snapshot matrix. In order to reduce the dimensionality, the first p number of most dominant basis functions are used, i.e., the first p eigenvectors that correspond to the p biggest singular values. All other basis functions are discarded. The first p POD modes are put together in the truncated basis matrix Φ_p. The truncated set of basis functions can still be used to reconstructed FOM solutions from the snapshot matrix, but since a lot of modes (with the smaller energies) are discarded, not all information is present and the reconstruction is not perfect. The fraction of the total system energy contained in the selected basis, η, is given by [7] as:

$$\eta = \frac{\sum_{i=1}^{p}\sigma_i}{\sum_{j=1}^{P}\sigma_j}, \tag{3}$$

where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_P \geq 0$ and where P is the total number of basis functions. The contained fraction of energy is a measure of how good truncated basis matrix was able to capture the characteristics of a system.

POD can be used in two ways: intrusive and non-intrusive. In intrusive methods the system equations are solved for the reduced POD basis. This requires modification of the FOM code, and thus the method is called intrusive. Methods that modify the code are called intrusive methods. Examples of intrusive methods are POD combined with Galerkin projection or with least squares Petrov Galerkin projection [7], [5].
Typical non-intrusive ROM methods utilize POD combined with interpolation in order to predict new solutions [7]. A disadvantage of POD based ROMs is that they fail to properly capture non-linear complex behaviour, e.g., advection dominated flow problems [8], [9], [5]. Neural networks could possibly solve this problem, for neural networks are able to approximate nonlinearities. This is a relatively new area of research. More can be read about ROMs in [10] and [11].

4.2 Neural Networks

4.2.1 Working principle

Neural networks come in many shapes and sizes. In order to understand the more complex networks, it is good to first take a look at the working principle of a simpler network. A simple neural network is the feedforward neural network, also known as the multi-layer perceptron (MLP). The general structure of a MLP is presented in Figure 1.

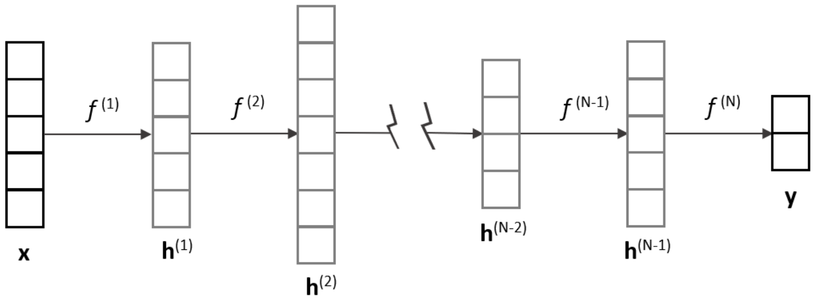


Figure 1: A multi-layer perceptron. The network consists of $N+1$ layers which all consist of multiple neurons / elements. The number of elements determine the dimension of the layer. $\mathbf{x}$ is the input layer of the network and $\mathbf{y}$ is the output layer. The hidden layers are $\mathbf{h}^{(i)}$ with i running from 1 to $N-1$. The functions $f^{(i)}$ with i running from 0 to N are mappings between layer $i-1$ and i, where it is used that $\mathbf{h}^{(0)} = \mathbf{x}$

The MLP consists of a number of layers which in their turn consist of a number of variables, the so-called neurons. The layer $\mathbf{x}$ is the input layer. The layer $\mathbf{y}$ is the output layer. All layers between the input and the output are hidden layers. The number of neurons per layer equal the dimensionality of that layer, e.g., in MLP of Figure 1, the input has a dimensionality of five, hidden layer one a dimensionality of five and the output a dimensionality of two.

The goal of the MLP (or any neural network) is to estimate a certain function $f^*(\mathbf{x})$. A network does so by mapping the input $\mathbf{x} \in R^m$ to output $\mathbf{y} \in \mathcal{R}^n$ via a function $f(\mathbf{x})$, which is actually a composite function from the different mappings between every layer. For the example of Figure 1, the total function is given as follows:

$$f(\mathbf{x}) = \left(f^{(N)} \circ f^{(N-1)} \circ \ldots \circ f^{(2)} \circ f^{(1)}\right)(\mathbf{x}), \tag{4}$$

where $f^{(i)}$ is the mapping from layer i to $i+1$ with $i = 1, \ldots, N$, with $N+1$ being the total number of layers where the first one is the input layer, the second one the first hidden layer and the last one being the output layer.

4.2.2 Layers

The mapping from layer to layer can be done in several ways. A classic method of layer mapping is done by the **fully connected layer**. The fully connected layer, a.k.a. dense layer, has a mapping which can be described as follows:
Suppose that a single fully connected layer, i.e., a MLP with no hidden layers, has input $\mathbf{x} \in \mathbb{R}^m$ and output $\mathbf{y} \in \mathbb{R}^n$. The mapping between input and output is then given as:

$$\mathbf{y} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{b}\right), \tag{5}$$

where the matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ is the weight matrix, where vector $\mathbf{b} \in \mathbb{R}^n$ is the bias vector and where the function g is the activation function. The value of a single output neuron y_j is determined by the activation and its input, which is the sum of every input neuron times an unique weight. For example, the value of the first output neuron is given as:

$$y_j = g\left(W_{i,j}\, x_i + b_j\right), \qquad i = 1, ..., m. \tag{6}$$

Notice that the first column of the weight matrix contains all the weights on which the first output neuron depends. This is the same for every other output neuron, except they use the row corresponding to their position. The second output neuron depends on the second row, the third output on the third row and so on. In other words, there is an unique weight for every multiplication in the fully connected layer. Note that the activation function g can either be visualised as being contained in the same layer as the linear mapping (as in Figure 2) or be visualised in a separate layer (as in Figure 5). A fully connected layer is visualised in Figure 2:

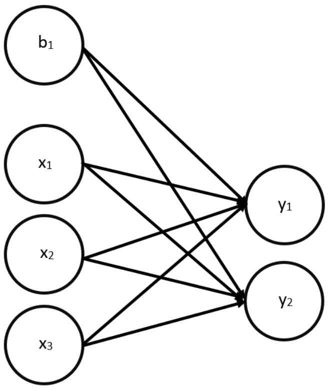


Figure 2: A simple fully connected layer, or dense layer. This is a typical way of visualizing such a layer. The input $\mathbf{x}$ is made up by 3 neurons and thus has a dimensionality of 3. The output $\mathbf{y}$ consists of 2 elements and thus has a dimensionality of 2. The neuron b_1 is the bias. Each output element is fed every input element plus the bias, all multiplied with an unique weight.

In this example, the input is of dimension $m = 3$ and the output is of dimension $n = 2$. The input to the activation function of every neuron is the sum of all incoming arrows. Every incoming arrow carries the value of the neuron or bias it originates from times a unique weight. This is a visualisation of the matrix multiplication from Equation (5).

If one chooses the activation function to be an identity function, i.e., $g(\mathbf{x}) = \mathbf{x}$, Equation (5) becomes a simple matrix multiplication and vector addition. This is a linear mapping. By adding multiple fully connected layers in series, all with an identity function or a similar linear function, the composite function of all layers together will be a linear one. That is, the total network function will be multiple matrix multiplications and vector additions, which are all linear operations. If however, one or more layers use a nonlinear activation function, the network will be able to create a nonlinear composite function which enables a better estimation of nonlinear functions.

Another way of mapping between two layers can be done with a convolution [12]. Layers whose output is created by a discrete convolution operation are called **convolutional layers**. In such a layer, one or more small weight matrices are slid across the input, multiplying their weights with the corresponding input values of the input map, outputting one or more output maps, referred to as feature maps. A mathematical description of the convolutional operation[1] (in two dimensions) is as follows:

$$S\left(i,j\right)=\sum_{p}\sum_{q}I\left(i+p,j+q\right)K\left(p,q\right), \tag{7}$$

where $S \in \mathcal{R}^{n_x \times n_y}$ is a (output) feature map, where $I \in \mathcal{R}^{n_x \times n_y}$ is the input tensor, or input feature map, and where $K \in \mathcal{R}^{k_x \times k_y}$ is the weight tensor, also known as the convolutional kernel or filter. Its dimension is given by k. i, j, p and q are indices. Note the Equation (7) is two dimensional and that the underscores of m, n and k indicate the size in a specific dimension. If the maps were to be flattened, the total size would be the product of the size of every dimension, i.e., $m = m_x \cdot m_y$. The two dimensional convolutional operation can easily be extended to three dimensions (by adding a third sum over the third dimension) or reduced to one dimension. In theory, the operation can be for whatever finite dimension one likes. The sums over p and q are performed over the elements of the kernel, i.e., they range from one to k_x and k_y respectively. For a common choice of convolutional kernel, a (3×3) sized one, p and q will run from one to three. The range of the indices i and j should only include the values for which there is an input value. This depends on the input tensor size alone. In order to show this, let a two dimensional convolutional kernel be of size k_x by k_y and let the input dimensions range from 1 to m_x and 1 to m_y, with m_x and m_y being positive. This means that the value of $i + p$ and $j + q$ should be between 1 and m_x and 1 and m_y respectively. Since $p_{\max} = k_x$ and $q_{\max} = k_y$, the output map S will be of size $(m_x - k_x + 1) \times (m_y - k_y + 1)$.

Whenever the convolutional kernel is not of size (1×1), the dimensions of the output feature map will be different to the dimensions of the input map. In order to preserve the input dimension, padding can be used. This mean that the input is extended during the convolutional operation. That is, the input dimensions will be increased from $m_x \times m_y$ to $(m_x + l_x) \times (m_y + l_y)$ where the padding sizes l_x and l_y should satisfy $l_x = k_x - 1$ and $l_y = k_y - 1$. The extra elements in the input tensor are appended around the tensor. They form a boundary layer around it. In the case of a one dimensional input of size n_x this means that $l_x/2$ elements are appended at start of the tensor and that $l_x/2$ elements are appended at the end of the tensor. The appended elements which are not part of the original input map are called the padding. The value of the padding can be set to anything, although the common choice is to set every padded element to zero, which is called zero padding. Another choice is to mirror the padding with the input map. In the one dimensional case, that would mean that the first right padding element will copy the value of the last element from the original input and the second right padding element will copy the second last element from the original input and so on.

The convolution operation can be extended by introducing an extra parameter called the **stride**, s. The stride is the step size of the convolutional kernel. By default, the stride is one, meaning a normal sliding of the kernel over the input map. In other words, every weight of the kernel is multiplied with an input element that is directly next to the element used in the previous slide step. In a strided convolution however, the two subsequent input elements that are multiplied with the same kernel weight in subsequent sliding steps, are a stride distance apart. That is, input

[1]An observant reader may notice that Equation (7) is actual the formula for calculating the 2D discrete cross-correlation of real signals and not the one for discrete convolution. The cross-correlation lacks the commutative property which convolution does have. However, this is not real important for a neural network and in order to be consistent with machine learning literature, the operation of Equation (7) will be referred to as a convolution.

element i and element $i+2$ instead of the default input element i and element $i+1$.

The use of strides means that a smaller number of total slide steps can be made over the input map since the slide steps are bigger. This in turn results in a reduced dimension of the output map, because the total number of slides determines the dimension of the output map:

$$n = \frac{m-k+l}{s} + 1. \tag{8}$$

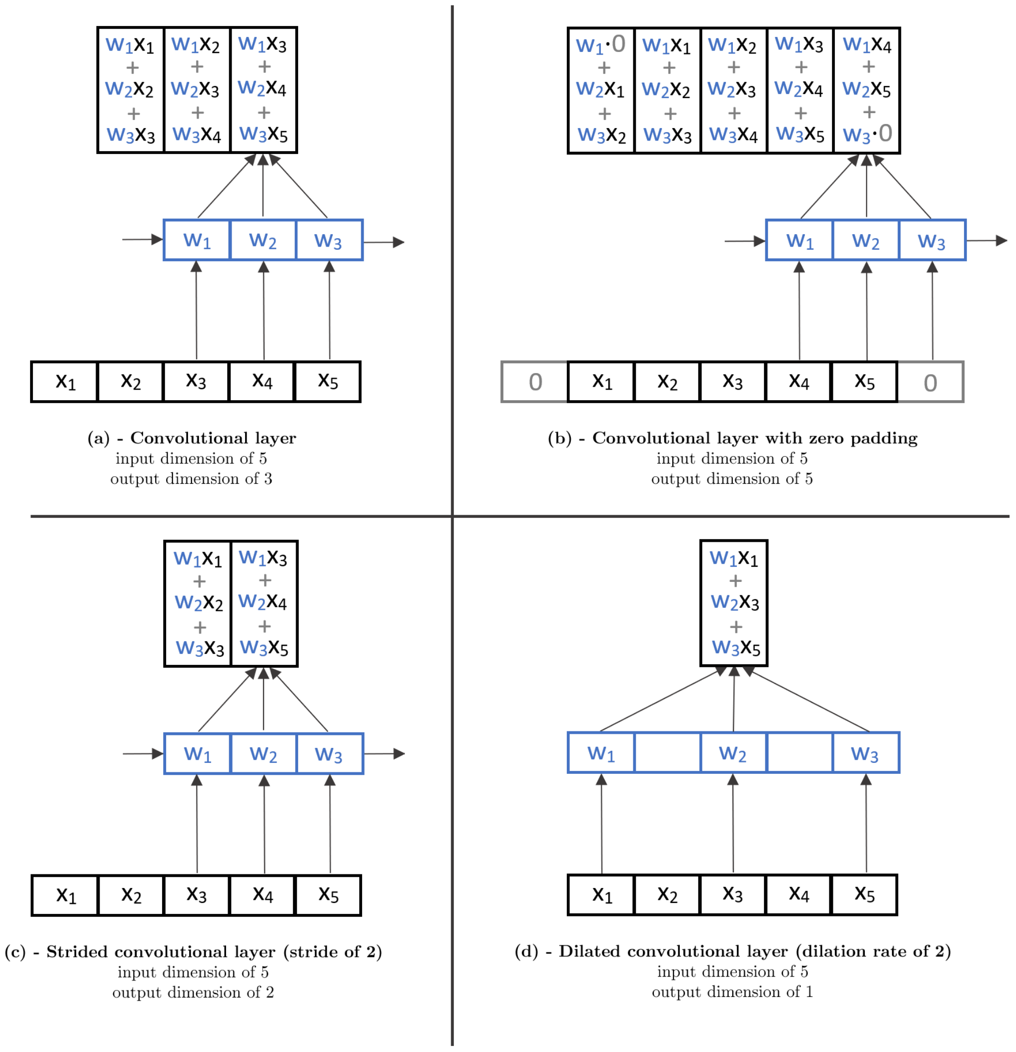
Equation (8) holds for every dimension, and hence, the subscripts indicating the dimension are omitted for brevity reasons. If one uses padding in order to get output dimensions equal to the input dimensions $n = m$, the padding is called 'same'.

Another extension to the convolutional layer is the use of dilated kernels, yielding a dilated convolution operation [13]. This simply means that elements which used to be directly next to each other in a default kernel, are now a distance d_{dil} away from each other. The distance d_{dil} is called the dilation rate and is one for a non-dilated kernel. The effective kernel size k now changes to $k_d = d_{dil}(k-1)+1$, where k is still the kernel size, but only in terms of the number of elements. The size of the field in the input map that is needed to determine a single output element has increased to a dimension of k_b for the input elements are now further apart. In other words, the receptive field of the kernel has increased due to the dilation. One could also choose to use a non-dilated filter of bigger size, e.g., a (7×7) kernel to achieve an increase in the receptive field, but that would require an increase in the number of weights.

Figure 4 shows the different type of convolutional operations for an one dimensional case. Combinations can be made. I.e., one could choose to use stride and padding or dilation and padding. In theory, a combination of dilation and stride could be possible, but is not supported by the Tensorflow framework.

If the mapping between two subsequent layers is done by a convolutional operation, one can have as many feature maps as one likes. The number of used kernels will scale with the desired number of output maps N_o. For a tensor with only one input map, the number of output feature maps is equal to the number of kernels. If however the input consists of N_i input feature maps, e.g., the input being the output of a previous hidden convolutional layer, the number of kernels will be the product of the number of input maps times the desired number of output maps. That is, a single output map is the sum of convolution operations over every input map, all convoluted with an unique kernel.

An advantage which a convolutional layer has over a dense layer is the reduced number of unique weights which need to be stored and optimized, i.e., a dense layer requires $m \times n$ weights whereas a convolutional layer requires $k \times N_i \times N_o$ parameters. Since m and n are usually quite big compared to k, a convolutional layer in general needs less parameters. For example, an input gray scale image of size 256×256 ($m = 65536$) to an output of 128×128 ($n = 16384$) will result in more than one billion parameters for a dense layer, where a convolutional layer with typical (3×3) kernels will only need $9 \cdot N_i \cdot N_o$ weight parameters. Here N_i equals one. If the input image were to be RGB, there would be three different input feature maps and hence N_i would be three. N_o is typically in a scale between ten and thousand. For hidden layers this means that N_i of the subsequent layer would be in that scale as well. This could still result in a big amount of weight parameter. In order to combat this, one could implement a convolutional layer where the number of output maps is lower than the number of input maps and use (1×1) kernels, as proposed in [14]. This layer serves as a reduction and should capture all necessary features from the input feature maps in a reduced number of output maps.



(a) - **Convolutional layer**
input dimension of 5
output dimension of 3

(b) - **Convolutional layer with zero padding**
input dimension of 5
output dimension of 5

(c) - **Strided convolutional layer (stride of 2)**
input dimension of 5
output dimension of 2

(d) - **Dilated convolutional layer (dilation rate of 2)**
input dimension of 5
output dimension of 1

Figure 3: Visualisation of 4 different types of convolution operations with the same input map of dimensionality 5 and the same sized kernel with weights w_1, w_2 and w_3. **(a)** - Default convolutional operation. the kernel can make 2 slides across the input meaning 3 different positions. This yields an output with a dimensionality of 3. **(b)** - Convolutional operation with zero padding. The kernel can calculate a value for 5 different positions, yielding an output dimensionality of 5. **(c)** - Strided convolution with a stride of 2. The kernel can slide only once before having at least one of its weights above a non existing input element. The two positions result in a output with a dimension of 2. **(d)** - Dilated convolutional operation. The kernel itself is dilated to a dimension of 5, meaning that there is only one possible position to calculate a value. This results in a output dimension of 1. In order to have an output dimension of 5, the input should be padded with 2 extra elements on both sides. Note that all output values have not yet gone through the activation function (i.e., not an identity function).

Another variant on the convolutional layer is the **transposed convolutional layer**. This layer is similar to the convolutional layer, except that the kernel works in the opposite direction. I.e., it takes only one input element to add a value to k different elements. The arrows in Figure 4 are

reversed. Padding is now used to remove boundary elements from the output instead of adding boundary elements to the input. If 'same' padding is used, boundary elements will be removed such that the output dimensions are equal to the input dimensions instead of being increased. The transposed convolutional layers can be used as upsampling layers, which means that the output maps will have an increased dimension compared to the input maps.
Besides the dense layer and the convolutional based layers, there are other types of layers which can be used as well, such as bilinear interpolation layers, pooling layers, batch normalization layers and layer normalization layers. Those will now be discussed, starting with bilinear interpolation layers.

A **bilinear interpolation layer** is an alternative layer used for upsampling a feature map to a higher dimension. It uses bilinear interpolation to find the value for the new neurons. It will first use linear interpolation to find the value in between two subsequent neurons in one direction of a feature map. This is also done for the two neurons directly above the other two. With the two interpolated values, one can perform another interpolation, but this time in the other direction. The found value did require linear interpolation on linear interpolated values, hence the name bilinear interpolation. The output is a smoothed out feature map with increased dimensions. This upsampling method does not require any additional model parameters to be saved.

A **pooling layer** should reduce the dimensionality. To this end, the input tensor will be divided into a grid with each grid point consisting of a number of elements. This grid point is called a pool. The amount of elements in the pool is determined by the pool size. An operation will be performed over all elements in one pool and the returned value will be the value of one output element. The most common operation is max pooling, which means that the maximum value in the pool will be returned. Another common one is average pooling where the average value of all elements in one pool will be returned. Pooling layers can be seen as a special type of convolutional layer where the kernel has unit weights and its size is equal to the pool size. In the case of max pooling, the activation function would be $g(\mathbf{x}) = \max(\mathbf{x})$ and in the case of average pooling it would be $g(\mathbf{x}) = 1/(\text{poolsize}) \times \sum_{i\in\text{pool}}(x_i)$.

Batch normalization layers perform a translation and a scaling operation on their input. To be more precise, every input feature map is translated to have a zero mean and is divided by its variance to have unit variance:

$$\hat{\mathbf{x}}_{i,c} = \frac{\mathbf{x}_{i,c} - \mu_{\mathcal{B},\mathbf{c}}}{\sigma^2_{\mathcal{B},c} + \epsilon}. \tag{9}$$

The intermediate output $\hat{\mathbf{x}}_{i,c}$ is a result of the standardization of a feature map of the original input. The i is an index indicating a single input of the total input batch. A batch is a set of multiple instances of inputs. i runs from 1 to B, the total number of instances in the batch. The second index, index c, runs from 1 to N_i and indicates which feature map is being standardized. $\mu_{\mathcal{B},c}$ is the average value of feature map c of all instances in the batch and $\sigma_{\mathcal{B},c}$ is the variance of the same set. ϵ is just a small positive scalar used for numerical stability. The real output is another translation and scaling:

$$\mathbf{y}_{i,c} = \gamma_c \hat{\mathbf{x}}_{i,c} + \beta_c \tag{10}$$

Here γ_c and β_c are learned scalar values. The translation and scaling are necessary for the optimization of a neural network. More about this in the Section 4.2.3. The recommended location of the batch normalization layer is after a linear operation (performed by either a dense layer or a convolution type of layer). This is recommended by the authors of the paper that proposed batch normalization layers [15]. Technically speaking, the term 'normalization' should be replaced with the term 'standardization', since a normalization operation will scale the inputs to be between 0 and 1 instead of being mean centered with unit variance.

Layer normalization layers are very similar to batch normalization layers as they too subtract the mean and divide by the variance. The major difference is that the mean and the variance are calculated on only the single input instance and not on the whole batch. I.e., the mean and variance in Equation (9) become μ_c and the variance σ_c respectively. Another smaller difference is the fact that the Equation (10) is not used and hence the intermediate output from Equation (9) is actually the final output. The layer normalization layer is invariant to scaling and shifting of the incoming weight matrix as long as the normalization layer is used after the dense layer (its linear part and before its activation function, if its a nonlinear one). The normalization layer was proposed by [16].

4.2.3 Optimization

Recall from Subsection 4.2.1 that the task of a neural network is to estimate a function $f^*(\mathbf{x})$. This function can be anything. It could be a classification function, that classifies whether emails are spam or not. The input are emails and the output is a likelihood. Another classification task could be an input image and an output returning a label for every pixel indicating of what object the pixel is a part of. Examples of these are the classification of each pixels from a ct-scan to the right class of tissue they represent. Or pixels from an image obtained by a self driving car which need to be classified to the type of vehicle or environment they represent. Another function type could be a regression task, e.g., the temperature prediction based on data from last week's weather. Classification tasks output discrete classes and regression tasks output continuous values. The function $f^*(\mathbf{x})$ is always correct. That is, it always returns the ground truth for any given input. The function is also often unknown. One can get close to the function $f^*(\mathbf{x})$ by estimating it with the composite function of a neural network. With the different types of layers discussed in Subsection 4.2.1, one can create a network whose composite function comes close to desired function. For the example of an object recognition function, a neural network can be constructed which is able to recognise the object present in the input image and correctly classify it with an accuracy of 99%. But here comes the tricky part. The neural network its composite function is determined by all its layer functions which in turn are all depended on their weights and biases. Hence, the composite function should be written as:

$$f_\theta\left(\mathbf{x}\right),$$

where the values of all the layer parameters are captured in the variable $\boldsymbol{\theta}$. There is an optimal value for $\boldsymbol{\theta}$ where the network is as close to the desired function $f^*(\mathbf{x})$ as possible. However, finding the optimal value for $\boldsymbol{\theta}$ is rather difficult. Namely, it is practically impossible to find the best performing values for the parameters by checking the performance of the composite function for every possible combination of values of $\boldsymbol{\theta}$. A viable approach is to optimize the model parameters via gradient descent.

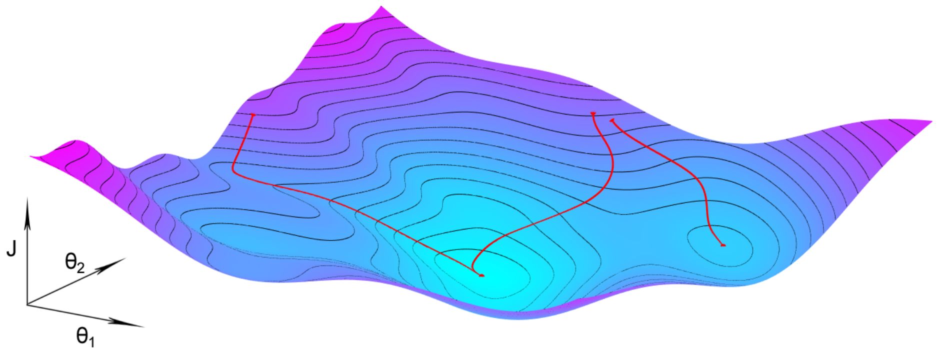


Figure 4: Gradient descent in a two dimensional loss landscape from three different starting points. The value of the loss $J(x, y|\theta_1, \theta_2)$ depends on the value of the two model parameters θ_1 and θ_2. The gradient descent algorithm steps downward through parameter space in the direction of the steepest decline till a local minimum is reached. The initial starting point influences the finish point.

Gradient descent is an algorithm which aims to find the minimum of a function by making small steps through the parameter space in the direction of the steepest downward gradient at each position till a (local) minimum is reached [12]. In order to use this algorithm for neural network optimization, one needs to define a function of which the minimum should be found. Since the aim is to create a composite function which is as close to the desired function as possible, an error function which quantifies and penalizes the difference between the two should work:

$$J(\boldsymbol{\theta}|\mathbf{x}) = L\left(f\left(\mathbf{x}|\boldsymbol{\theta}\right), f^*(\mathbf{x})\right). \tag{11}$$

The function J is called the loss function or cost function. L can be a selection of functions who are desired to be continuous and differentiable. Two common choices are the mean squared error and the mean absolute error (although the mean absolute error is not-differentiable in zero) in regression and generative tasks. Unfortunately, Equation (11) will not work for gradient descent, for $f^*(\mathbf{x})$ is unknown and thus not enough input argument can be fed to the loss function. What one can do however, is gather multiple instances of known input and output, i.e., $\left\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}_{\text{true}}\right\}$ of the black box $f^*(\mathbf{x})$ and put those instances together in a batch, so $\mathbf{y}^{(\mathbf{i})}_{\text{true}} = f^*(\mathbf{x}^{(\mathbf{i})})$. If B number of $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$ sets are gathered and put in a batch, it can be used as a substitute for the unknown function, changing Equation (11) into:

$$J(\boldsymbol{\theta}|\mathbf{x}, \mathbf{y}_{\text{true}}) = \frac{1}{B}\sum_{i=1}^{B} L\left(f_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right), \mathbf{y}^{(i)}_{\text{true}}\right), \tag{12}$$

where $f_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)$ can also be written as $\mathbf{y}^{(i)}$ and where the term $1/B$ is used to calculate the mean value of L. If a smaller set of data points is, the upper limit of the sum is changed and so should the $1/B$ term, in order to steal determine the mean. The cost function in Equation (12) has known inputs and can be minimized with gradient descent. Several variants on the gradient descent algorithm exist, but the most straight forward one is the stochastic gradient descent (SGD) algorithm, shown in Algorithm 1.

The algorithm calculates an update for the model parameters for every mini-batch. A mini-batch is a subset of size b taken from the total batch of data of size B. It estimates the gradient numerically and updates the weights by stepping in the opposite direction of the gradient. The size of the step is determined by the gradient estimate as well as the learning rate α_t. This hyper parameter could be a constant value, but it is better if it follows a custom scheme. The descent stops when a proper low loss has been found or when it seems that the model is not learning anything anymore.

Algorithm 1 Stochastic gradient descent

```
Require: initial value $\boldsymbol{\theta}$
Require: learning rate schedule $\alpha_t$
  $t \leftarrow 1$
  while stopping criteria not met do
      Set learning rate $\alpha_t$ according to the schedule
      Sample mini-batch of sets of $\left\{\mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \ldots, \mathbf{x}^{(b)}, \mathbf{y}^{(b)}\right\}$ from the total batch of size $B$
      Compute the gradient estimate $\hat{\mathbf{g}} \leftarrow \frac{1}{b} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right), \mathbf{y}^{(i)}\right)$
      Update the parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_t \hat{\mathbf{g}}$
      $t \leftarrow t + 1$
  end while
```

This requirement is quantified in the stopping criteria. More about the schemes and the stopping criteria in Subsection 4.3. Note that algorithm 1 should be called mini-batch gradient descent if $b > 1$ and gradient descent if $b = B$.

SGD updates the parameters every step with only the values of the gradient estimate at that specific point. If the next position in parameter space is situated on a saddle point or in a local minimum, the estimated gradient of that new point will be close to zero which results in a very small step in parameter space. During the next iteration, the estimated gradient will again be close to zero since the position in parameter space has hardly changed and is still in a small gradient area. This yields another small step which in turn results in another small gradient estimate for the next iteration. The loss does not decrease anymore and is said to be stuck in a local minimum. Luckily, the fact that every iteration is performed on another mini-batch, will lessen this effect, for the loss landscape will not look exactly the same for new data. I.e., every mini-batch yields a slightly different loss landscape and thus different gradients for the same value $\boldsymbol{\theta}$. Still, improved gradient descent algorithms exist to prevent these effects from happening and help to improve the overall learning speed.

Several variants exist. They extend the SGD algorithm with a momentum term, an adaptive term or a combination of both. A momentum term takes the previous gradient estimates into account when updating the parameters. That is, the gradient estimate in algorithm 1 does update a momentum variable which in turns updates the parameters. If the gradient descent has been stepping downward in one direction for some time, the momentum variable has increased for that direction and hence, encountering a gradient estimate which is in a completely different direction will not cause the parameter update to be exactly in that direction since the momentum still contains the value for the original direction it was travelling in. Also, the momentum term prevents the gradient descent algorithm to stall when it encounters near zero gradients. Several options exist for the momentum. SGD with momentum updates the momentum variable in each step with the gradient descent. The momentum can depend in different ways on the gradient step. One can add up all gradient estimates or let the influence of older gradient estimates fade away to zero. The adaptive term means that learning rate of every parameter is unique and has its own calculation. The Adam optimizer [17] is adaptive and has a exponentially decreasing momentum term and even a second momentum term. It has proven to be a successful gradient descent variant and is often used for training.

One thing all gradient descent based algorithms have in common is the need for the estimate of the gradient. The gradient is computed numerically by the so-called back-propagation algorithm. **The back-propagation algorithm** [18] returns the derivatives of the parameters at their current position. It does this efficiently by making use of the chain rule. Recall that the overall function of a neural network, $\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$ is a composite function build up by all functions between the layers. Suppose that one wants to determine the derivatives of all parameters of a simple

MLP network. The network consists of sequence of layers, namely the input $\mathbf{x}$, $(N-1)$ hidden layers $h^{(i)}$ and the output layer y. The hidden layers are split up into a linear mapping from $h^{(i)}$ to the intermediate layer $z^{(i)}$ and the mapping from the intermediate layer with the activation function to the next layer $h^{(i+1)}$. The linear mapping of layer i is done with function $f^{(i)}_{\boldsymbol{\theta}^{(i)}}(\mathbf{x})$ and the activation function of the same layer is done with $g^{(i)}$. The parameters of every linear layer function, $\boldsymbol{\theta}^{(i)}$, are put together in the total parameter tensor $\boldsymbol{\theta}$. See Figure 5 for a more visualised presentation of the network.

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "f^{(1)}"] & \mathbf{z}^{(1)} \arrow[r, "g^{(1)}"] & \mathbf{h}^{(2)} \arrow[r, "f^{(2)}"] & \mathbf{z}^{(2)} \arrow[r, "g^{(2)}"] & \mathbf{h}^{(3)} \arrow[r, "f^{(3)}"] & \dots \arrow[r] & \mathbf{z}^{(N-1)} \arrow[r, "g^{(N-1)}"] & \mathbf{h}^{(N)} \arrow[r, "f^{(N)}"] & \mathbf{z}^{(N)} \arrow[r, "g^{(N)}"] & \mathbf{y}
\end{tikzcd}$$

Figure 5: A neural network with only forward connections with one input layer, one output layer and $N-1$ hidden layers. The numbering of the hidden layers starts with 2 up to N instead of 1 to $N-1$ in order to make the back-propagation more intuitively. Every layer is split into the linear operation from $\mathbf{h}^{(i)}$ to $\mathbf{z}^{(i)}$ with $f^{(i)}$ and the activation from $\mathbf{z}^{(i)}$ to $\mathbf{h}^{(i+1)}$ with $g^{(i)}$.

The total composite function of the network can written as follows:

$$\mathbf{y} = g^{(N)}\left(f^{(N)}\left(g^{(N-1)}\left(f^{(N-1)}\left(g^{(N-2)}\left(\dots f^{(2)}\left(g^{(1)}\left(f^{(1)}\left(\mathbf{x}\right)\dots\right)\right)\right)\right)\right)\right)\right),$$

which itself is used in the cost function given in Equation (12). The notation of the composite function is not a very concise notation, but it does show the dependencies of every function. The partial derivatives of the cost function with respect to the parameters of the last hidden layer can be found via the chain rule. To illustrate: The value of element i of the output is given as

$$y_i = g^{(N)}\left(z_i^{(N)}\right), \tag{13}$$

where the input to the activation function is only element $z_i^{(N)}$ since the activation function operates element wise. The value of $z_i^{(N)}$ depends on the linear mapping of the last hidden layer its input $\mathbf{h}^{(N)}$ and the parameters $\boldsymbol{\theta}^{(N)}$:

$$z_j^{(N)} = f^{(N)}_{\boldsymbol{\theta}^{(N)}}\left(\mathbf{h}^{(N)}\right). \tag{14}$$

The influence of the change in a single weight $\boldsymbol{\theta}_k^{(N)}$ on the cost function is the partial derivative of the cost function to this weight. Here, the index k steps from 1 to the total number of parameters used by function $f^{(N)}$. By using Equation (13) and Equation (14) together with the chain rule, the obtained partial derivative is as follows,

$$\frac{\partial J}{\partial \theta_k^{(N)}} = \frac{\partial J}{\partial L}\sum_j \frac{\partial L}{\partial y_j}\frac{dy_j}{dz_j^{(N)}}\frac{\partial z_j^{(N)}}{\partial \theta_k^{(N)}}, \quad j = 1, \dots, \dim(\mathbf{y}), \tag{15}$$

where $\frac{\partial J}{\partial L}$ is found via Equation (12) and where $\frac{dy_j}{dz_j^{(N)}}$ is equal to $\frac{dg^{(N)}(z_j^{(N)})}{dz_j^{(N)}} = g^{(N)\prime}(z_i^{(N)})$. In order to find the partial derivative of the cost function of any parameter k of any arbitrary layer l, one needs to know how a change in the parameter influences every output element $h_j^{(l+1)}$. This is done via the chain rule in similar fashion as in Equation (15) and is as follows,

$$\frac{\partial h_j^{(l+1)}}{\partial \theta_k^{(l)}} = g^{(l)\prime}\left(z^{j\,(l)}\right)\frac{\partial z^{j\,(l)}}{\theta_k^{(l)}}, \quad j = 1, \dots, \dim(\mathbf{h}^{(l+1)}). \tag{16}$$

In order to know how the change in layer $l+1$ will influence the cost function, the partial derivative of the cost function with respect to the layer $l+1$ should be known. This too can be found with the chain rule for the partial derivative of a layer with respect to its previous layer is simply a multiplication of the derivative of the activation function at with values $\mathbf{z}^l$ and the derivative of the linear mapping in the previous layer with respect to its input values:

$$\frac{\partial h_j^{(l+1)}}{\partial h_k^{(l)}} = g^{(l)\prime}\left(z_k^{(l)}\right)\frac{\partial z_k^{(l)}}{\partial h_k^{(l)}}. \tag{17}$$

The gradient can now be obtained. One inputs a single data instance, a mini-batch or the total batch to the network. With this input, the linear operation of layer 1 can be calculated. The results yield the value for $\mathbf{z}^{(1)}$. Those values can be passed forward into the activation function of layer 1 yielding the input of layer 2. This process of forwarding the input data all the way to the output is called the forward pass. After the forward pass, all layer inputs $\mathbf{h}^{(l)}$ and intermediate layer inputs $\mathbf{z}^{(l)}$ are known. The cost function and its derivative with respect to the model parameters can be calculated with the values of the right-hand side of Equation (16) and Equation (17), for they are known. That is, the derivative of every activation function is known, and after the forward pass, all values of the intermediate layers are known as well. The derivative of an intermediate layer to its parameters, $\frac{\partial z^{j\,(l)}}{\theta_k^{(l)}}$, depends solely on the values of the input layer, $\mathbf{h}^{(l)}$, whose values have been found during the forward pass. The derivative of the intermediate layer to its input, $\frac{\partial z_k^{(l)}}{\partial h_k^{(l)}}$, depends solely on the layer parameters, $\boldsymbol{\theta}^{(l)}$. Those values have been found during the forward pass as well. It can easily be shown why an output derivative with respect to the input values depends on the layer parameters and why the output derivative with respect to the layer parameters depends on the layer input. Let the output of Equation (6) be a single element and let the activation function be an identity mapping. Then, the derivative of the output with respect to its input values are the weights W_i and the derivative of the output to the weights are the inputs x_i. For the very first step of the learning process, i.e., the first step of a gradient descent variant, the model parameters should have a initial value and those can be chosen more or less arbitrarily.

4.3 Heuristics of neural networks

Subsection 4.2.1 and Subsection 4.2.3 give the basics about the working principle and the learning process of neural networks. Unfortunately, the actual practice of creating and training a network is not a total exact process, but involves a lot of trial and error. Luckily, there are several known heuristics for several decision that need to be made during the process. This subsection provides an overview on those heuristics.

4.3.1 Over-fitting and under-fitting

The task of a neural network is to approximate a unknown function from data. The datasets are used in the training process during which the model parameters are adjusted such that the error between the true and predicted output data is minimized. In other words, the model is trained to optimally fit the training data and not to optimally approach the unknown, but desired function. Luckily, since the data is related to the unknown function, there is a good possibility that the fit which the network makes for the training data is a close approximation of the unknown desired function. One can check this by splitting the data in a training and a test set. The model should not see the test data before the training has finished. If the error of the test set is approximately equal to the error of the training set at the end of the training (slight deviation is possible, but at least the same order of magnitude), one can say that the model generalizes well. If however, the error on the test set is much worse than the error on the training set, the model overfits the training data. That is, it fits the data so accurate that it fails to generalize. Two possible solutions are to increase the size of the training data batch or to add a regularization term to the loss function. Adding a regularization term to the loss function enforces uniqueness to the weights. That is, where there were several possible sets of parameters at a local minimum, there is now a smaller set, namely those with a smaller norm. The activation functions which follow the linear mappings are now more likely to receive an input closer to zero since small weights are preferred. Increasing the training batch comes with the cost of gathering data which is often costly. Another possible solution is to reduce the complexity of the model. A reduction in complexity means that the network will have a harder time to specify its function to a degree where it starts over-fitting by estimating every data point of the training set correctly, but that it will rather find a more general function. The big drawback to this approach is that the network may become to simple where it can no longer make a good enough of a fit. This effect is called under-fitting. To determine if one's model is under-fitting, the predictions should be plotted and checked or an upper bound for the error should be set, which has to be satisfied by the model on both the training and test set. A validation set can be used during the training itself to monitor if over-fitting occurs. This is done by checking the losses of both the training and validation set every certain amount of gradient descent steps. The model only adjusts its parameters with data from the training set. If one observes that the training loss goes down but that the loss of the validation data starts to go up, the model has passed its optimum and is starting to overfit. A stopping criteria for the training can be made by monitoring the training and validation losses. This can be done via a learning curve, which is a plot showing both the training and validation loss as a function of the training process. The process can be quantified by the number of gradient descent iteration or the number of data epochs. A data epoch is a single pass through the all data of the training batch.

A good practice is to **standardize** the input data, that is, the mean value should be subtracted and that result should be divided by the variance. This standardization operation is given in Equation (9), and is the key operation in batch normalization layers. The batch of which the mean and variance should be calculated is the training batch. The validation and test batches should be transformed with those values and not their own mean and variance values. Standardization is strongly recommended since data which is close to zero (which is the case for standardized data) usually results in faster convergence [18]. Also, it is better to have inputs of the activation of a output to be of both signs. That is, some outputs should be positive and some negative. If they all have the same sign, the parameters will all be updated into the same direction, all will either increase or decrease. This will result in a fluctuating motion during the optimization.

4.3.2 Weight initialization and batch sizes

The initial step of the gradient descent algorithm requires an initial value for the model parameters. This is done during the weight initialization. The starting point does matter in the gradient descent algorithm. To see this, imagine a two dimensional loss landscape with a peak in the middle and a local minimum on opposite sides (left and right) of the peak. If the starting position is on the left slope of the peak, gradient descent will ascent further downwards on the left side of the peak and find the local minimum on the left, but it will find the right local minimum if the starting point is on the right side of the peak. In other words, starting at different points in parameter space will most likely result in different end points. However, this does not have to be a problem, if both local minimum yield a similar composite function. In practice, this seems to be the case [12]. There may exist several combinations of parameter values which all result in a similar composite function. The real task is to initialize the weights in such a way that the network will converges to a sufficient low local minimum and that the process is as fast as possible. There is no best way of doing this, but symmetric weights (e.g., all with the same starting value) should always be avoided for those weights result in a symmetric parameter update which makes training impossible or will at least slow down the progress significantly. Setting the weights to large values isn't a good thing either for the computed gradient by the back propagation will scale with the weight value which can result in exploding gradients, where the gradient steps accumulate to bigger and bigger steps. The training will become divergent. A common way of initializing the weights is to set their value with a random distribution around zero.

The mini-batch size is the hyper-parameter indicating the number of data points used for a single parameter update during the gradient descent progress. If a single data point is used, the back propagation will return the gradient of the current position on the loss landscape which is depended on only the values of the single data point. For the next gradient descent step, another single data point will determine the shape of the loss landscape. One can imagine that the shape of the landscape will fluctuate a lot between different data points. This will also result in fluctuations in the gradient which in turn will translate in a noisy training. Another disadvantage of SGD is the fact that batch normalization layers will estimate the mean and the variance incorrectly (for the subset is too small), which will greatly reduce their effectiveness. In the other extreme case, where the whole batch is used to determine the loss, the back propagation will always determine the gradient in the same loss landscape, and thus the convergence will be smoother. The disadvantage of this approach is that the network needs to forward pass the whole training batch for only one gradient descent step. The optimum is found in the mini-batch approach where the batch size is not too small in order to get a noisy descent, nor is it too big that unnecessary extra data points are forward passed. For small mini-batch sizes the batch normalization layers will still perform rather poorly. One could consider using layer normalization layers or group normalization layers instead in that case.

4.3.3 Activation functions

Activation functions are responsible for the nonlinear mapping ability of a neural network. Important properties of activation functions are if they are bounded or unbounded and if they are differentiable. A bounded activation function has the advantage that the outputs are always within the bounds and so will be the activations. This means that the total bias shift is bounded which is good for the learning progress, since smaller bias shift results in faster learning. The disadvantage of bounded activations however is the fact that the gradient goes to zero the closer it gets to its bounds. A very small gradient translates into a very slow learning. A popular zero-mean bounded activation function is the hyperbolic tangent function.
A popular choice for an unbounded (or rather, only bounded below) nonlinear activation function is the rectified linear unit (ReLU) [19]. This function returns zero for all negative inputs and is identity for all positive inputs.

$$g(z) = \max\{0, z\}. \tag{18}$$

The derivative of the ReLU is a straightforward, but non-differentiable in zero:

$$\frac{dg(z)}{dz} = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0. \end{cases} \tag{19}$$

Technically speaking, the derivative does not exist in zero, but one can chose to set it to the left-hand limit, the right-hand limit or the average of those two. The computation of both the ReLU function and its derivative is not as heavy as other common activation functions, since it does not require computations of any exponentials, logarithms or powers, but only needs to perform a single max comparison for the ReLU and a multiplication for the derivative. Another advantage of the ReLU is the fact that it promotes sparsity. That is, if the inputs are less than zero, the output neuron will be a true zero.
A big additional advantage of ReLU is the fact that it does not have the problem of vanishing gradients since its gradient is always one for positive inputs. The zero gradient of negative inputs has the disadvantage that it could cause dead neurons. Those are neurons which are never used during the training. This can happen since a negative input to the ReLU yields a zero gradient, which can result in the gradient descent algorithm not updating the associated parameters. Since ReLU is zero or positive, the mean of the output of every subsequent layer is more prone to a positive mean shift. This mean shift, or internal covariate shift [15], slows down training and hence it is suggested by [15] to use their proposed batch normalization layers after every linear mapping layer in order to shift the mean back to zero and the variance back to one. This enhances the training of models and is often used. The only downside to this layer is the fact that it is sensitive for small mini-batch sizes for it cannot estimate the mean and variance of the whole batch very

well and the found values may vary too much between two mini-batches. A possible solution would be the use of layer normalization layers, as proposed by [16]. However, the authors found that the layer normalization does not seem to improve the performance for convolutional layers and suggest that more research should be done on that topic.

There exist proposed variants on ReLU which do not have the problem of dead neurons, but they all sacrifice the sparsity advantage ReLU does have. One of the proposed variants is the leaky ReLU (LReLU) [20], which has the same linear mapping as ReLU for positive inputs, but has a negative mapping of $g(z) = \alpha z$ where $0 < \alpha < 1$. By not having $\alpha = 1$, the LReLU will be nonlinear like the ReLU, but it will have a gradient of α instead of zero for negative inputs.
Another variation on the ReLU is the ELU, or Exponential Linear Unit [21]. This activation function as defined as

$$g(z) = \begin{cases} z, & z > 0 \\ \alpha\left(e^x - 1\right), & z < 0. \end{cases} \tag{20}$$

with its derivative being as follows

$$g(z) = \begin{cases} 1, & z > 0 \\ \alpha e^x, & z < 0, \end{cases} \tag{21}$$

where in the case of $\alpha = 1$, the derivative is continuous. The ELU has the advantage that its mean output is closer to shifted closer to zero than the ReLU. According to [21], ELU performs better than ReLU and LReLU for networks deeper than five layers. Also, they found that batch normalization did not enhance the performance of ELU, whereas it does for ReLU.
The popular choice for activation function has shifted from the sigmoid to the tanh to the ReLU function. It is suggested by [12] that activation functions which are close to linear have a faster convergence than activation functions which are less linear. This is confirmed by experiments done by [22] where ReLU outperformed the tanh activation functions.

4.3.4 Learning Rate

Selecting the right learning rate is crucial for the training progress of a neural network. If the learning rate is too big, the training becomes unstable and the optimization progress could diverge. This is easy to see by recalling that step size of the parameter update is proportional to the learning rate. A big learning rate results in a bigger step through the loss landscape in parameter space. The bigger the step, the bigger the chance of overstepping into a different regime with a totally different gradient. The updates will fluctuate more through parameter space with increasingly bigger steps for the weights are more likely to become bigger during every update, which results in bigger steps, and so on. Consequently, the loss will grow instead of decrease and the training fails.

Choosing an extra small learning rate will prevent those problems, but comes with the disadvantage of having a higher chance of getting stuck in a critical point, far above an acceptable loss value. That is, when the gradient descent algorithm steps onto a critical point, the gradient will approach zero and the parameter update will be a rather small one. If the learning rate has a big enough value, it may counter this small multiplication and the gradient descent can escape the critical point. If the critical point is a plateau where the gradients are just very low, a low learning rate will escape just like an optimal learning rate, but it will take more steps, and hence more computational steps. Also, depending on the stopping criteria, the training may stop prematurely. The slowed down learning effect actually applies not only the plateau phenomena, but to any place in the loss landscape. The overall learning process of a lower than optimal value learning rate will take more time.

Learning with the optimal learning rate is preferred, but finding the optimal learning rate is hard, especially for every different location in parameter space. A good heuristic is to start the training with a bigger learning rate (order of magnitude of 10e-3) and to gradually decrease its value as the training proceeds. Two ways of doing this are by a custom schedule or by reducing the learning rate based on the loss improvement. That is, by monitoring the validation and training loss, one can see when the loss stagnates for a certain learning rate. When the stagnation occurs, the learning rate should be decreased. A typical decrease is the halving of the learning rate. Stagnation is measured by the number of steps or epochs in which no better loss has been found, where a better loss may only be considered better when its value is a certain percentage or absolute value below the current best loss instead of just being lower. If no better value is found for a set amount of steps, the learning rate will decrease. The amount of steps is called the patience. By setting the patience too low, it could happen that the learning rate is reduced while it is slowly learning on a loss plateau. The last thing that would benefit the learning on such a plateau is the decrease in learning rate. Determining the right value for the patience is a case of trail and error and is problem specific.

If one has trained a network multiple times for different minor adjustments in the network and/or the hyper parameters, one has more experience and can write a custom schedule for the learning rate which indicates after how many steps the learning rate should decrease. Reducing the learning rate via a predetermined schedule learns faster but is less robust.

4.3.5 Regularization

A common practice in neural networks is the incorporation of regularization of the activations or the biases or the weights or a combination of those. The regularized cost function will be as follows,

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{\lambda}{\dim(\boldsymbol{\theta})}\Omega(\boldsymbol{\theta}), \tag{22}$$

where λ is a positive scalar and $\Omega(\boldsymbol{\theta})$ is a function penalizing the value of the weights. The biases should not be penalized. It is common to penalize the value of the weights by the the L1 or L2 norm, which is the sum of the absolute values and the sum of the squared values respectively. By optimizing the regularized cost function, low values for the weights are preferred. This helps to prevent the over-fitting of noise in the training set, which would result in the failure the generalize.

4.4 Network architectures

4.4.1 Residual blocks

A simple network is one where every layer is followed by another layer, all the way till the output layer. VGG models [23] do have this type of architecture. However, the deeper the network becomes, i.e., the more subsequent layers, the harder the optimization. An important building block which enabled better training for deep networks residual blocks [24], which are blocks of convolutional layers in series combined with one skip connection from the input directly to the output of the block. The addition is followed by an activation.

Suppose one has a simple model of a few sequential layers, with input $\mathbf{x}$ and composite function $H(\mathbf{x})$. During the optimization process, the network has to learn the correct output of the network, which is $H(\mathbf{x})$. The difference between the input and the output is given as $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$ and is called the residual. By moving the input term to the left-hand side and subsequently mirroring the sides, one obtains a new equation for the output,

$$H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}. \tag{23}$$

In order to let the proposed simple network have such a composite function like Equation (23), a skip connection from the input to right before the output should be added. The network becomes a residual block, whose basic structure is shown in Figure 6. The residual function can be a series of multiple layers of different types. The original paper itself recommends using two or more linear layers, where every linear layer, except for the last one, is followed by a batch normalization layer and a activation layer. The final output dimensions and number of feature maps of the sequential layers in the block should be equal to the input dimensions and number of feature maps since those are added together. Different dimensions and number of feature maps can be compensated for with extra padding and/or 1×1 convolution layer.

The advantages of residual blocks is the fact that the the skip connections prevent the gradient from vanishing for deeper layers, and deeper networks with higher level of abstractions can be trained. If the optimal output of a residual block is an identity mapping, the residual which has to be learned, should be zero. [25] found that the learning process was more efficient for networks with residual blocks than for similar networks without residual blocks.

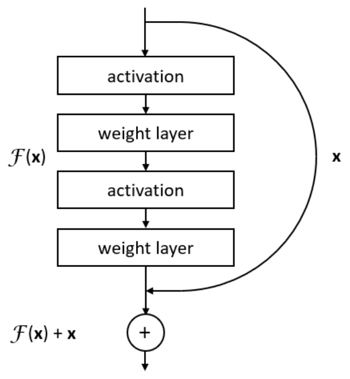

activation
weight layer
$\mathcal{F}(\mathbf{x})$
activation
weight layer
$\mathbf{x}$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
+

Figure 6: The general structure of a residual block. The input is split into a skip connections which immediately maps to the addition operation and into a branch which encounters activations, batch normalization and weight layers before being input to the addition operation. The batch normalization layers are not shown, but if used, they are recommended to be put between the activation and the weight layer.

Note that the general residual block structure shown in Figure 6 is a variant of the original proposed block [24], namely an improved version proposed in [26]. They empirically showed better results for a network where the activations happen before the weight layers and where the skip connection can stay as close to identity as possible, i.e., the skip connection does not encounter any activation layer.

4.4.2 Inception blocks

Another block structure is the inception block [27]. A big motivation behind the inception block was to increase the computational efficiency. I.e., increase the width and depth of a network for a constant value of parameters. This is accomplished by running the input parallel through different types of convolutional filters (different kernel sizes) and a pooling layer. 1×1 convolutions are used to reduce the number of input feature maps in order to reduce the total number of computations. They are sometimes called the 'bottleneck layer', since they downscale the number of feature maps to the lowest number of intermediate feature maps in the block. The outputs of all paths are concatenated together before the output. The motivation for the use of different sized convolutional kernels along the different parallel path is the fact that different kernels capture different characteristics of input maps.
Variants on the inception block have been proposed where a skip-connection is incorporated in

the block, creating a residual block variant, as in [28]. A general structure of an inception block can be seen Figure 7.

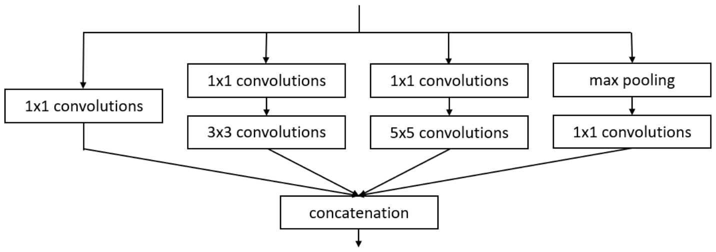


Figure 7: The inception block as proposed in [27]. The 1×1 bottle neck blocks output different number of feature maps for every branch.

4.4.3 Autoencoders

Autoencoders are a special type of unsupervised neural networks. Unsupervised means that there is no labeled output associated with the input data. An AE does not need this, since it should recreate the input at its output. It does so by first reducing the input to a space of reduced dimensions. This reduction is done by the encoder $\mathcal{E}(\mathbf{x})$. The encoder's output is the reduced space and is called the latent space with dimension d, where usually $m >> d$. The output has to be reconstructed from those d latent variables by the decoder $\mathcal{D}(\mathbf{x})$. The decoder's output is of the same dimension as the input of the encoder. The total composite function is given as:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathcal{D}\left(\mathcal{E}(\mathbf{x})\right) \tag{24}$$

The loss function should penalize the difference between the input and the output. In this way, the AE is really forced to find model parameters in such a way that all essential information is saved within the latent variables. [29] suggests inserting a few linear dense layers right before the latent space to enhance the performance of the model. The encoder and decoder can be used separately to encode and decode to and from the latent space respectively.

AEs can be used for several applications, the most common being the reduction of the dimensionality of data. If a linear decoder is together with a mean squared error loss function, the AE is learning a similar subspace representation as the POD mentioned in Section 4.1. The strength of AE reduced dimension representation is in the nonlinear activation functions. The AE is able to approximate nonlinearities better.

Variants on the AE exist. One of them is the **Sparse AE** where the loss function is extended with a term which penalizes the numbers of active latent variables. This way, a latent variable will only be activated if it is really needed, which forces the sparse AE to represent the high dimensional input in as few latent variables as possible.
Another variant on the AE is the **Variational AE**. The latent variables of this variant are given as probability distribution instead of deterministic values.

A visual representation of the generic layout of an AE can be seen in Figure 8:

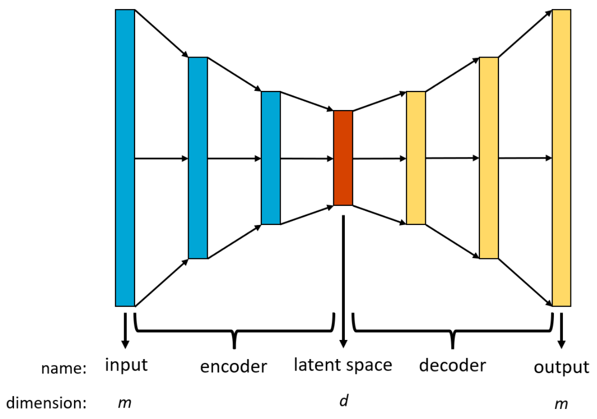


Figure 8: The generic layout of an autoencoder. The input is fed forward through the encoder till it reaches the latent layer, or the latent space. The latent variables are fed forward through the decoder to the output. The ideal output equals the input. The encoder and decoder can both be used separately after training.

If one or more of the linear layer operations in an AE consist of convolutional operations then the AE can be called a convolutional autoencoder. The bigger the input to an AE, the more favourable a CAE in terms of model parameters, since convolutions need less parameters than dense layers.

4.5 Neutron diffusion problems

For this thesis, a ROM will be created for physics problem. There are many physics problems, but the ones to which the CAE will be applied to are all neutron diffusion problems, which form a part of nuclear reactor physics.

To elaborate more on reactor physics [30]; In a nuclear reactor, a nuclear reaction is sustained. This reaction is measured by the neutrons that fly around inside the reactor. If a neutron collides with the fissile material (which is the fuel inside a reactor), there is a chance that nuclear fission will occur in the fissile nuclide. This fission reaction yields, a.o., new neutrons. If those neutrons happen to collide with another fissile nuclide and cause a nuclear fission, a new generation of neutrons will be created, which in turn can cause new fission reactions and create a new generation of neutrons. Controlling the neutron distribution inside of a reactor is a necessary requirement for (safely) sustaining a nuclear reaction. The neutron distribution of an arbitrary volume is described with the neutron transport equation, which can also be expressed in the angular neutron flux $\varphi(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t)$, which is just the angular neutron density multiplied by the neutron velocity. Describing the neutron transport equation in terms of the angular neutron flux is more convenient, and reads as:

$$\begin{aligned}\frac{1}{v}\frac{\partial\varphi}{\partial t} + \hat{\mathbf{\Omega}}\cdot\nabla\varphi + \Sigma_t(\mathbf{r},E)\varphi\left(\mathbf{r},E,\hat{\mathbf{\Omega}},t\right) = \\ \int_{4\pi} d\hat{\mathbf{\Omega}}' \int_0^\infty dE'\Sigma_s\left(E'\to E,\hat{\mathbf{\Omega}}'\to\hat{\mathbf{\Omega}}\right)\varphi\left(\mathbf{r},E',\hat{\mathbf{\Omega}}',t\right) + Q\left(\mathbf{r},E,\hat{\mathbf{\Omega}},t\right).\end{aligned} \tag{25}$$

The first term on the left-hand side is the change of the angular neutron flux of a specific energy E, with a specific direction $\hat{\mathbf{\Omega}}$ on a specific location $\mathbf{r}$ at a specific time t. The second term covers all neutrons that move away from the specific location. The third term on the left-hand side covers the change of the angular neutron flux at a specific location, direction, energy and time due to interactions with the medium it is in. All possible interactions with the medium are covered in the total macroscopic cross-section Σ_t. The total cross-section is a summation of the following macroscopic cross-sections:

- Σ_s, the macroscopic scatter cross-section. This indicates the probability of a neutron to undergo a scatter collision per unit length of travel. In a scatter operation, both the direction and the energy of the neutron can change. A further distinction can be made between elastic and non-elastic scattering cross-sections.
- Σ_c, the macroscopic capture cross-section. This is the probability of being captured by an atom nucleus per unit length of travel.
- Σ_f, the macroscopic fission cross-section, which quantifies the chance of a neutron to be captured by a nuclide atom and to cause a fission of that nucleus per travelled unit length.

All cross-sections have the unit of 1/cm and depend on the material of the medium and the energy of the neutron. The capture and the fission cross-section together add up to the total absorption cross-section Σ_a.
The double integral on the right-hand side of Equation (25) quantifies the number of neutrons from any direction and energy to undergo a scattering reaction in such a way that their new energy and direction will be equal to the specific energy and direction on the left-hand side. The Q term is the source term which can be further specified, depending on the problem. An external neutron source can be described as well as the neutron generating fission reactions, which depends on the angular flux itself.

The neutron diffusion equation can be obtained from the neutron transport equation. This is done by integrating the transport equation over all possible angles and by subsequently using the diffusion approximation where $\int \hat{\mathbf{\Omega}} \cdot \nabla\varphi d\Omega = -\nabla \cdot D(\mathbf{r}, E)\nabla\varphi$. The neutron diffusion equation is as follows:

$$\begin{aligned}\frac{1}{v}\frac{\partial}{\partial t}\varphi(\mathbf{r}, E, t) = \\ &\nabla \cdot D(\mathbf{r}, E)\nabla\varphi(\mathbf{r}, E, t) - \Sigma_t(\mathbf{r}, E)\varphi(\mathbf{r}, E, t) + \\ &Q(\mathbf{r}, E, t) + \int_0^\infty \Sigma_s(\mathbf{r}, E' \to E)\varphi(\mathbf{r}, E', t)\, dE',\end{aligned} \tag{26}$$

where $\varphi(\mathbf{r}, E, t)$ is the scalar neutron flux per energy. $D(\mathbf{r})$ is the diffusion coefficient and is a material constant which depends on the cross-sections of the material.

Two common variations on the neutron diffusion equation both have to do with the energy dependency. The continuous energy value can be assumed to all be the same value, effectively eliminating the energy dependency. This is called the **one group diffusion equation**. Equation (26) becomes

$$\frac{1}{v}\frac{\partial}{\partial t}\varphi(\mathbf{r}, t) = \nabla \cdot D(\mathbf{r})\nabla\varphi(\mathbf{r}, t) - \Sigma_a(\mathbf{r},)\varphi(\mathbf{r}, t) + Q(\mathbf{r}, t). \tag{27}$$

The integral in Equation (26) vanishes since all neutrons have the same energy and thus no neutrons can enter the group of energy E from another energy value E' by colliding with the material. The neutron flux becomes independent from the energy.
The other common variant on the neutron diffusion equation is the **multigroup neutron diffusion equation.** In order to obtain this multigroup form from Equation (26), one has to discretize the continuous energy E into multiple discreet energy groups E_g. The multigroup equation becomes:

$$\frac{1}{v_g}\frac{\partial}{\partial t}\varphi_g(\mathbf{r}, t) = \nabla \cdot D_g(\mathbf{r})\nabla\varphi_g(\mathbf{r}, t) - \Sigma_{tg}(\mathbf{r})\varphi_g(\mathbf{r}, t) + \sum_{g'}\Sigma_{g'g}\varphi_{g'}(\mathbf{r}, t) + Q_g(\mathbf{r}, t), \tag{28}$$

where $\varphi_g(\mathbf{r}, t)$ is the total neutron flux of energy group g. All material parameters are now with subscript g, indicating their average value in that energy group. The summation term over all energy groups on the right-hand side contains the matrix element $\Sigma_{g'g}$ which is the average value for the scattering cross-section of neutrons in energy group g' to energy group g.

As mentioned earlier, the source term Q can be further specified. An external source of neutrons can be added which can be at whatever position and of whatever energy one likes, at least in theory. In practice, spontaneous fission of nuclides in a material can be used as an external source. The neutron flux does not have influence on the activity of the external neutron flux.
A neutron source which does depend on the total neutron flux is the fission reactions of fissile nuclide within materials, such as Uranium-235. If a fissile nuclide captures a neutron, there is a change that a fission will occur. This will yield ν new neutrons on average, depending on the material and the energy of the captured neutron. The chance of a fission reaction to happen is dependent on the neutron flux and the fission cross-section of the material. Assuming that both the external and the neutron induced source types are present within an arbitrary volume, the source term of the one group problem reads

$$Q\left(\mathbf{r},t\right)=\nu\Sigma_f\left(\mathbf{r}\right)\varphi\left(\mathbf{r},t\right)+Q_{\text{ext}}\left(\mathbf{r},t\right), \tag{29}$$

where $Q_{\text{ext}}\left(\mathbf{r},t\right)$ quantifies the external source. The source term for the multigroup source is as follows

$$Q_g\left(\mathbf{r},t\right)=\chi_g\sum_{g'}\nu_{g'}\Sigma_{fg'}\left(\mathbf{r}\right)\varphi_{g'}\left(\mathbf{r},t\right)+Q_{\text{g,ext}}\left(\mathbf{r},t\right), \tag{30}$$

where the summation over g' describes the amount of neutrons of energy group g created by a fission caused by neutrons from energy group g'. The energy distribution of the created neutrons is quantified by the energy distribution χ_g.

5 Methodology

5.1 Method

The goal of this master thesis project is to construct a ROM model based on a CAE (CAE based ROM) for a physics problem with FOM parameters $\boldsymbol{\mu} \in \mathcal{R}^q$. A dataset of FOM solutions of different values of $\boldsymbol{\mu}$ will be acquired by a finite element approach in MATLAB [31]. Both the FOM parameters and the solutions will be saved in a dataset, which will be split into a training, validation and test set in a 70%, 15% and 15% fashion respectively. The FOM solutions are used to train a CAE. The FOM parameters are used to create a multivariate polynomial regression model from the FOM parameters to the latent space of the CAE. The joint model of the polynomial regression model and the decoder part of the CAE is the CAE based ROM. Its performance will be compared to a non-intrusive POD based ROM. The POD is calculated on the same training set with which the CAE has trained. The motivation behind comparing the CAE based ROM with a POD based ROM is the fact that a POD is linear and a CAE is not (given that nonlinear activations are used). Since the CAE is able to better approximate the nonlinear manifold on which the intrinsic solution lays, one would expect the CAE based ROM to outperform the POD based ROM in accuracy.

The procedure to create a CAE based ROM for a physics problem consists of the following steps:

1. Acquire FOM solution data.
   The FOM solution data is found via a finite element solver for different values of $\boldsymbol{\mu}$. The values are drawn from a probability distribution over the range of their possible values. The acquired solution data should be split into a train, validation and a test set.

2. Construct and train a CAE.
   A CAE has to be constructed and trained on the acquired FOM solutions. The training is done with the training set and validated with the validation set. The construction and training is the main challenge of the master thesis.

3. Construct and train a multivariate polynomial regression model for the FOM parameters to latent space variables.
   The FOM parameters are the input to the regression model and the latent variables the output. The FOM parameters are known and the corresponding latent space variables are found by inputting the FOM solution into the encoder. Different regression models can be chosen, however in this master thesis project a multivariate polynomial regression model will be used.

4. Create the ROM by joining the linear regressor and the decoder.
   The input of the ROM are FOM parameters and the output is the CAE output, which is as close to the FOM solution as possible.

A similar procedure is used for the POD based ROM with which the CAE based ROM will be compared:

1. Acquire FOM solution data.
   This is the same data as used for the training and evaluating of the CAE based ROM

2. Perform POD and create a truncated basis.
   The POD is performed on the training set, just like the CAE training. The number of POD modes in the truncated basis matrix should be equal to the latent space dimension for comparison.

3. Construct and train a linear regressor for the FOM parameters to POD coefficients.
   Similar to the procedure of the CAE based ROM, a linear regression model to go from the FOM parameters to the POD coefficients needs to be constructed. Here, a multivariate

polynomial regression model will be used of preferably the same order as the one used in the CAE based ROM in order to have a fair comparison.

4. Create the ROM by joining the linear regressor and the decoder.
   The input of the ROM are FOM parameters and the output is the POD prediction based on the truncated number of POD modes.

The two ROMs are visualized in Figure 9. It is expected that the encoder and decoder of the CAE will capture all nonlinearities of the physics problem they are applied on which will result in a smooth latent space. The smoother the latent space, the better the polynomial regression from step 3. This holds for the POD based ROM as well. One can gain insight in the (indirect) influence of the FOM parameters on the latent variables by plotting the value of the latent space variables as a function of FOM solutions, created by different values of one or two different FOM parameters. All other FOM parameters should be set to a constant.

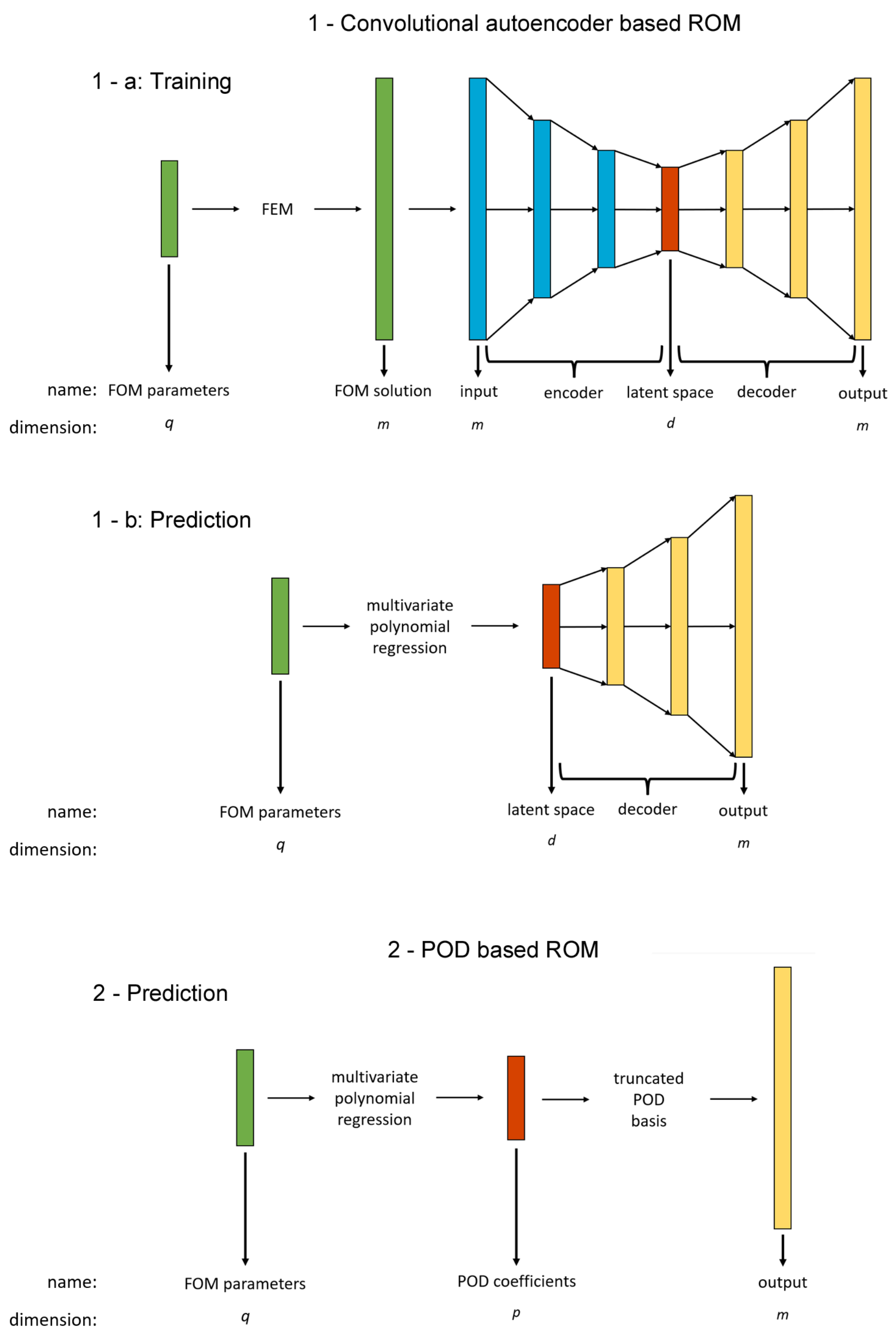


Figure 9: The two different ROMs which will be used. (1a) - The CAE based ROM. The CAE will be trained for FOM solutions for different values of the FOM parameters $\boldsymbol{\mu}$. (1b) - After the training, the ROM is created by putting together a simple multivariate polynomial regression model and the decoder of the CAE. (2) - The POD based ROM predicts the solution for a set of FOM parameters via a multivariate polynomial regression model and the truncated POD basis. This basis is obtained with the same FOM solutions used in the training for the CAE based ROM, but instead of training on this set, a POD operation is performed over it.

The main focus of the research will be the creation and training of the CAEs. A ROM will be created for the best performing CAE. The CAE creation and training, as well as the POD operation and ROM construction will be done in Python [32]. The library used for the machine learning is the Tensorflow library [33]. The training is performed on GPUs provided by the web IDE 'Google Collab'. Exact details about the GPU are unknown and during every session another GPU is given access to which means that exact speed comparisons are not possible between different models. The total FOM data will be distributed in 70% training set, 15% validation set and 15% test set.

5.2 Application examples

The physics problems for which a CAE will be constructed are three variants of the time-independent neutron diffusion problem. The first problem is a one group neutron diffusion problem with cross section perturbations, the second one a multigroup diffusion problem with cross section perturbations and the third one is another one group diffusion problem but with perturbations in the geometry. The last problem is expected to be the hardest to capture with a small truncated set of POD modes.

5.2.1 One group neutron diffusion with cross-section perturbations

The first problem is a time independent one group neutron diffusion with cross-section and external source perturbations. The external source is of constant value and is present in all fuel material. All FOM parameters are listed in Table 1. Their values range from 50% to 150% of their unperturbed values. During the data acquisition the FOM parameters are drawn from an uniform distribution over their allowed range of values. There is a total of 8 adjustable FOM parameters for this problem. The used geometry is shown in Figure 10. Dirichlet boundary conditions apply on the outer edges.

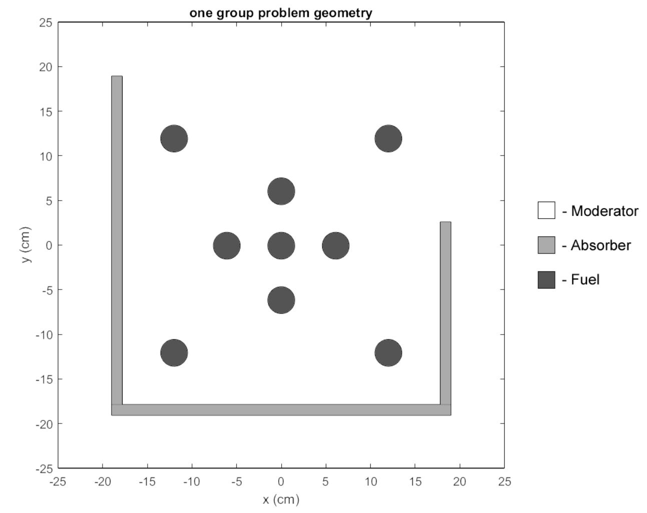


Figure 10: The used geometry for the one group problem with cross-section perturbations. The geometry consists of three different materials. The three rectangles forming the thin U-shape are 'absorber' material, which does not have a fission cross-section and has a relatively high capture cross-section. All circles in the geometry are made up by 'fuel' material, which have a fission cross-section and an external source. This can be seen as a constant spontaneous decay happening inside the material. The big square in which the circles and rectangles lay is the third material and is the 'moderator' material. The moderator material has a relatively high scatter cross-section and does not have a fission cross-section. Dirichlet boundary conditions apply on the outer edges.

	'mat' subscript letter	Scatter cross-section $\Sigma_{s,\mathrm{mat}}$ (cm^{-1})	Capture cross-section $\Sigma_{c,\mathrm{mat}}$ (cm^{-1})	Fission cross-section Σ_f (cm^{-1})	External source per unit surface $Q_{\mathrm{ext,f}}/A$ (neutrons cm$^{-2}s^{-1}$)
Fuel	f	$4.08 \cdot 10^{-2}$	$9.2 \cdot 10^{-3}$	$2.70 \cdot 10^{-2}$	10
Moderator	m	$8.05 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$		
Absorber	a	$5.20 \cdot 10^{-2}$	0.2945		

Table 1: FOM parameters that can be adjusted in the one group problem. The unperturbed values are shown. The perturbed values range from 50% to 150% of the unperturbed values. Note that although neutrons cannot scatter from one energy range to another one for this one group problem, the scatter cross-section is still perturbed since it influences the diffusion coefficients with the elastic scattering.

5.2.2 Multigroup problem with cross-section perturbations

The second physics problem is a time independent multigroup neutron diffusion problem, where the geometry consists of two materials, namely fuel and moderator. The FOM parameters, all of which can be perturbed from 50% to 150% of their unperturbed value, are again the cross-sections and the external source term, except that the cross-section are subdivided into the different energy groups. All unperturbed values are shown in Table 2, together with the upper and lower bounds of the different energy groups. The used geometry is shown in Figure 11.

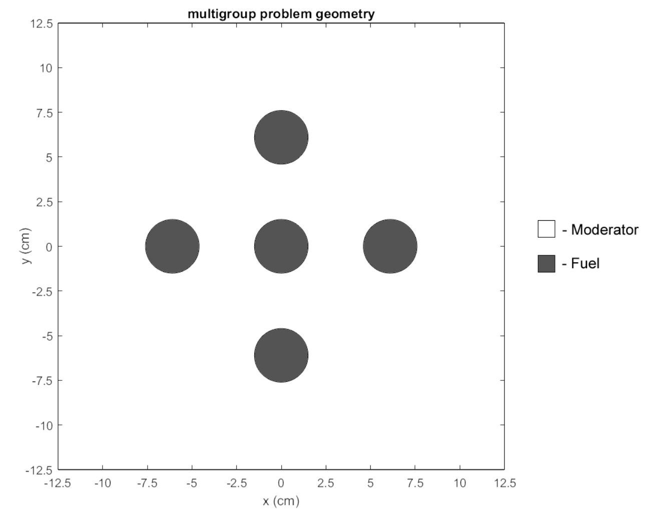


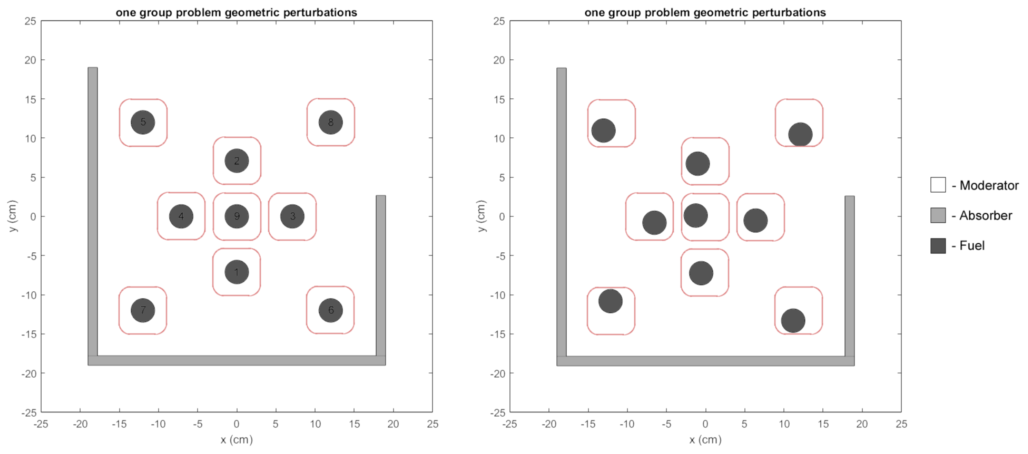
Figure 11: The used geometry for the multigroup problem. Two different materials are present in the geometry. All circles are made up by 'fuel' material which has fission cross-sections and a constant external fission source for every energy group. The square in which the fuel pins lay is 'moderator' material and has no fissile material or external sources in them. Dirichlet boundary conditions are applied on the edge for the total flux of every energy group.

	Energy group g	$E_{upper\, bound}$	Scatter cross-section $\Sigma_{sg,mat}$ (cm^{-1})	Capture cross-section $\Sigma_{cg,mat}$ (cm^{-1})	Fission cross-section $\Sigma_{fg,f}$ (cm^{-1})	External source per unit surface $Q_{g,\mathrm{ext},f}/A$ (neutrons cm$^{-2}s^{-1}$)
Fuel (subscript $mat = f$)	1	20 MeV	0.1699	$8.13 \cdot 10^{-4}$	$7.20 \cdot 10^{-4}$	1
	2	8.2 MeV	0.3261	$2.90 \cdot 10^{-3}$	$8.19 \cdot 10^{-4}$	1
	3	1.2 MeV	0.4536	$2.03 \cdot 10^{-2}$	$6.50 \cdot 10^{-3}$	1
	4	82 keV	0.4581	$7.77 \cdot 10^{-2}$	$1.86 \cdot 10^{-2}$	1
	5	450 eV	0.2818	$1.22 \cdot 10^{-2}$	$1.78 \cdot 10^{-2}$	1
	6	6 eV	0.2839	$2.82 \cdot 10^{-2}$	$8.30 \cdot 10^{-2}$	1
	7	0.18 eV	0.2816	$6.68 \cdot 10^{-2}$	0.216	1
Moderator (subscript $mat = m$)	1		0.1586	$6.010 \cdot 10^{-4}$		
	2		0.4130	$1.610 \cdot 10^{-5}$		
	3		0.5900	$3.373 \cdot 10^{-4}$		
	4		0.5824	$1.900 \cdot 10^{-3}$		
	5		0.7123	$5.700 \cdot 10^{-3}$		
	6		1.239	$1.500 \cdot 10^{-2}$		
	7		2.613	$3.720 \cdot 10^{-2}$		

Table 2: Unperturbed FOM parameters for the multigroup problem. There are 6 different parameters for every energy, which with 7 energy groups, yields 42 FOM parameters. All parameters can be perturbed from 50% to 150% of their unperturbed values. The upper and lower bound of every energy group is given. Note that $\Sigma_{sg,f}$ and $\Sigma_{sg,m}$ are the total scatter cross-section of an energy group g of the fuel material and the moderator material respectively. The total scatter cross-section of a material of an energy group Σ_{sg} is the summation of $\Sigma_{g'g}$ over g'. The external source is given per unit surface and has the same strength for every energy group within the fuel and is given per unit surface. The total external source strength is determined by the total surface of the fuel, A, in the geometry. The moderator material does not have any external source nor does it have any fissile nuclides.

5.2.3 One group problem with geometric perturbations

The last physics problem for which a CAE will be created is a one group problem like the one in Subsection 5.2.1, except instead of cross-section perturbations, the location of the fuel pins / circles will be perturbed. The range of the x- and y-location of the fuel pins is plus and minus the radius of its unperturbed location. The radius of the fuel pin is 1.5 cm. Figure 12 shows the unperturbed locations and all possible locations. Since there are 9 fuel pins in total, there are 18 FOM parameters, which are the 9 x-coordinates and the 9 y-coordinates of the fuel pins. The material cross-sections and the external neutron source are set to the constant value equal to the unperturbed values as the proposed problem in Subsection 5.2.1.


one group problem geometric perturbations
one group problem geometric perturbations
x (cm)
y (cm)
- Moderator
- Absorber
- Fuel

(a) - unperturbed geometry (b) - random geometric perturbations

Figure 12: The one group problem with perturbations in the fuel locations. The left sub figure shows the default locations together with the domain with possible locations per fuel pin (the red squircle around every pin.) The right sub figure shows a geometry for which all FOM parameters have a random value picked from a uniform distribution of their possible x and y values.

5.3 Proposed network structure: parallel residual block

In order to create a CAE that can approximate the neutron diffusion problems, the following block is proposed: The parallel residual block.
As the name suggests, the block consists of two branches of stacked convolutional layers which are parallel to each other. Both layer stacks have their own skip connection. The first layer stack has default 3×3 convolutional layers. The second branch however starts its stack with a dilated convolutional layer. The dilated stack will pick up different characteristics than the non-dilated stack. Due to both layer stacks having their own skip connection, the layers can be seen as two residual blocks which operate parallel to each-other. Their outputs are added together right before the output of the parallel residual block (or parallel resBlock). A schematic overview of the parallel resBlock is shown in Figure 13. The motivation for this block is two-fold: Firstly, residual blocks learn faster, and secondly, different receptive fields capture different information of the input. This is the reason behind the dilated filter in the second branch. One could use a bigger kernel instead of a dilated filter, but this method of increasing the receptive field comes with the cost of extra bigger kernels and thus more weight parameters and computations. A similar block to the parallel residual block has been proposed by [34], but it did not include the dilated filters.

The parallel residual block will be used in the encoder part. In the decoder part, a transposed version will be used. This version uses the transposed convolutional layers instead of convolutional layers. Also, the order of filters will be reversed for every branch. In other words, the dilated filter which is the first convolutional layer in branch 2 will be the last (transposed) convolutional filter of the transposed block.

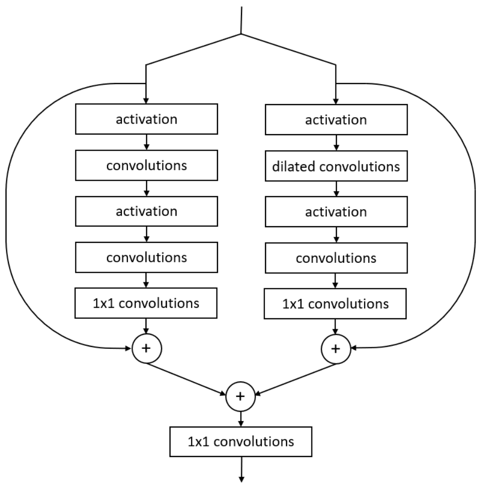


Figure 13: The structure of a parallel resBlock. The first filter in the second branch is dilated to change the receptive field with respect to the first branch. 1×1 convolutions are used to change the number of output maps of both branches such that they equal the amount of feature maps from the skipped connection. The last 1×1 layer changes the number of output maps to the desired number if requested. All non 1×1 convolutional kernels are 3×3 kernels. All convolutional operations are of 'same' zero-padding.

5.3.1 Used models

For every one of the three aforementioned problems a different type of CAE will be constructed. The first problem, the one group diffusion problem with cross-section perturbations will be approached with a VGG-like autoencoder. The second problem, the multigroup problem with cross-section perturbations, will make use of residual blocks. The last problem, the one group problem with geometric perturbations will use both parallel residual blocks and normal residual blocks.

Figure 14 shows the fraction of contained energies per number of POD modes used. The highest shown number of modes is the first number of modes that together hold more than a fraction of 0.999 of the total energy. The one group problem with cross-section perturbations only needs two modes to cross this threshold. The solutions to this problem seem to be almost captured by only the scaling of the first POD mode. One could say the same for the one group problem with geometric perturbations, since the energy fraction contained in only the first mode is even bigger. This will appear to not be the case, as can be expected, since the geometric perturbations do not cause a big difference in the solutions magnitude, but more in the overall shape.

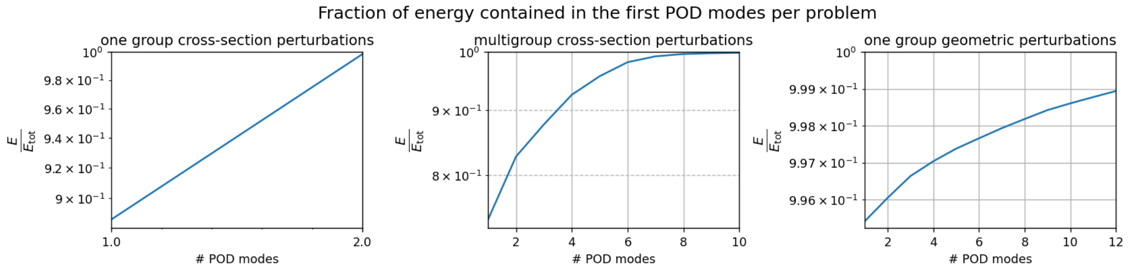


Figure 14: The fraction of the total energy contained within the first number of POD modes for the three different diffusion problems. The highest number of modes in every subplot is the first time that the fraction of total energy exceeds 0.999.

6 Results

In this section, results on the three diffusion problems will be shown, as well as a brief mention of some extra results with concern to the network design and optimization.

6.1 Diffusion problems

All problems have FOM solutions of $128 \times 128 \times$ nGroups, which means that a single input for both one group problems is of dimension $m = 16384$ and that a single input to the multigroup problem is of dimension $m = 114688$, since it consists of 7 groups. All input data is standardized to be zero-centered and of unity variance. This is performed once for every energy group in the multigroup problem in order to have all groups within the same scale.

6.1.1 One group problem with cross-section perturbations

A simple CAE has been created for this problem. There are only forward connections and no skip connections. The input meets three consecutive convolutional layers with 'same' zero padding, before being halved in dimensions by a max pooling layer. This happens four times in total leaving the feature maps with a dimension of 8×8. At this dimension level, three more convolutional layers are encountered before two nonlinear and three more linear dense layers are met. The last linear dense layer maps to the latent space which has a dimensionality of $d = 4$. This concludes the encoder. The decoder is built in similar fashion to the encoder, except that instead of five dense layers, only one has been used. Also, all convolutional layers are replaced with transposed convolutional layers and the max pooling layers are replaced with bi-linear interpolation layers. There is one extra feature to the network and that is that the first convolutional layer of the encoder and the second-last convolutional layer of the decoder consist of two parallel convolutional filters of which one is non-dilated and the other is dilated. The motivation behind this is that the non-dilated filter will capture different feature than the dilated filter. The last layer of the decoder is used for feature map reduction to the required number of output feature maps, which is one. All nonlinear activations are 'elu' functions. No normalization layers have been used. A schematic overview of the used model can be seen Figure 15. In short, the encoder is 15 convolutional layers and 5 dense layers deep, which gives the encoder a total of 1.3 million parameters. The dimension is reduced from $m = 16384$ to $d = 4$. The decoder is 16 convolutional layer (of which one is a feature map reduction layer) and 1 dense layer deep, giving the decoder 1.3 million parameters in total. This brings the number of parameters used in the CAE up to a total of 2.6 million.

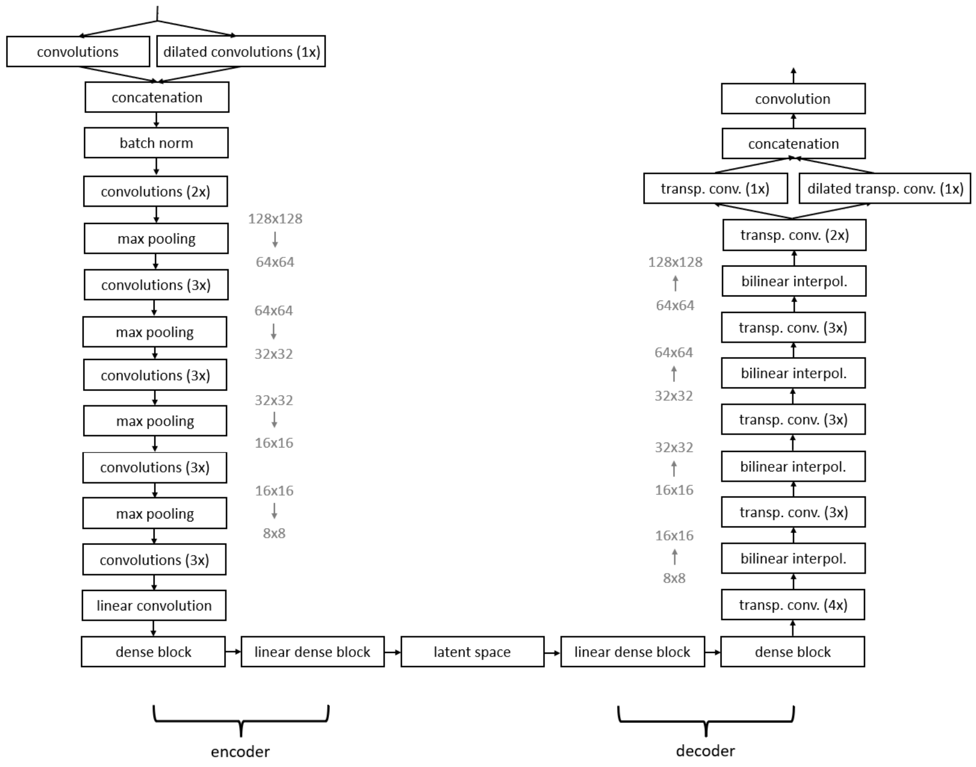


Figure 15: The used CAE network for the one group problem with cross-section perturbations. If the kernel size of a convolutional layer is not mentioned specifically, a 3×3 kernel has been used. Every block with *convolutions (x)* and *transp. conv. (x)* are subsequent convolutional layers and subsequent transposed convolutional layers respectively. The *(x)* indicates the number of subsequent layers. All nonlinear activations are 'elu' functions.

The training is performed on 6300 samples with the Adam optimizer and the learning rate is reduced if a stagnation in the validation loss is detected. The stopping criteria is based on stagnation in the validation loss as well, except for a longer patience. After the training, a multivariate polynomial regression model is used to approximate the latent variables from the FOM parameters. In order to gain more insight in the behaviour of the latent variables, the values of the latent variables are plotted against the perturbations of a single and of dual perturbations of FOM parameters. Such a plot of the single FOM perturbation influence can be seen in Figure 16, together with the influence of the same FOM parameter on the POD coefficients. The performance of both multivariate polynomial regression models is given in Table 3 and the performance of both ROMs is shown in Table 4. The given errors are equal to the errors of the CAE and the POD projection. In other words, the FOM parameters to latent variables regression does not add significant error to the prediction of the decoder. Two predictions of the CAE based ROM and the POD based ROM are shown in Figure 17, where one predictions is on the datum[2] with the highest mean squared error when predicted by the CAE based ROM and the other one is the datum with the highest mean squared error when predicted by the POD based ROM. Two random validation datum predictions can be found in the Appendix, Figure A.1. Note that the CAE (and the CAE based ROM) predicts worse for data points with a lower magnitude. This can be traced back to the use of the mean square error as loss function during the training. The same relative error in a data point with a higher magnitude will yield a bigger contribution in the loss than the same relative error of a data point with a lower magnitude. During training, minimization of the loss is

[2]Datum is the singular form of data and is an alternative word for data point.

the goal, and hence the data points with bigger magnitude have more influence.

	Training set	**Validation set**	**Test set**
FOM parameters to latent vars			
mse $\times 10^{-3}$	2.9	6.1	7.1
FOM parameters to POD coef.			
mse $\times 10^{-9}$	1.1	2.4	2.4

Table 3: Mean squared error of the multivariate polynomial regression from the FOM parameters to the latent space and to the POD coefficients.

	Training set	**Validation set**	**Test set**
CAE based ROM			
mae $\times 10^{-2}$	6.6	6.9	7.1
mse $\times 10^{-2}$	1.0	1.1	1.3
POD based ROM			
mae $\times 10^{-2}$	3.0	3.0	3.0
mse $\times 10^{-2}$	2.9	2.9	3.0

Table 4: Mean absolute error (mae) and mean squared error (mse) of both the CAE based ROM and the POD based ROM for the one group problem with cross-section perturbations. All errors indicate the difference between the prediction and the ground truth values.

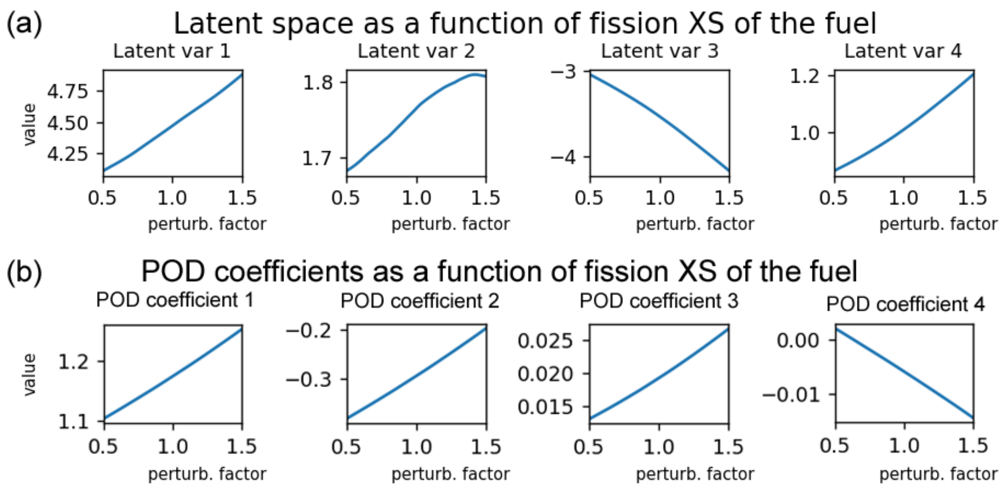


Figure 16: The influence of the perturbation of the fission cross-section of the fuel on (a) - the latent variables, and (b) - the POD coefficients. The fission cross-section is perturbed from 50% to 150% of its unperturbed value. All other FOM parameters are kept unperturbed.

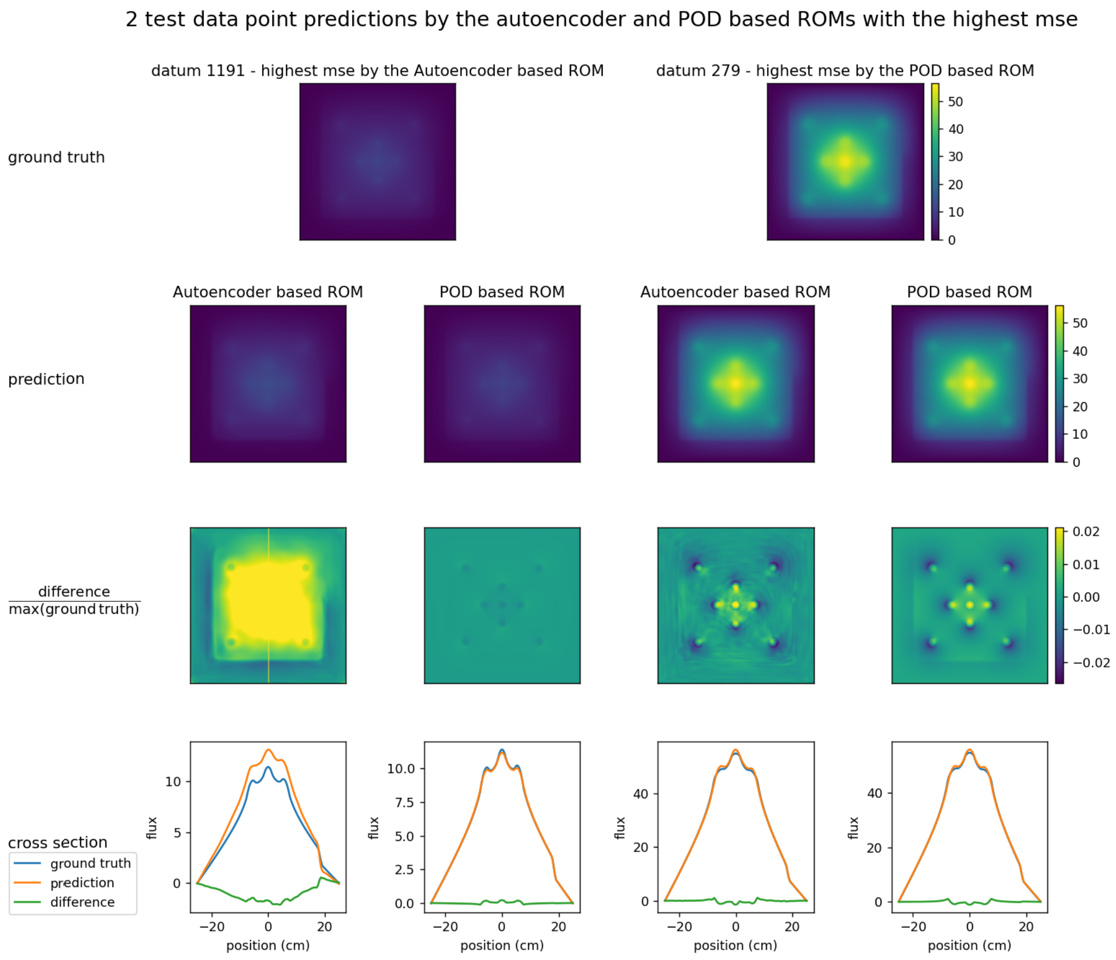


Figure 17: Subplots of the prediction, difference and vertical intersection through the geometry of two data points from the test set of the one group problem with cross-section perturbations. The first data point gave the highest mse when predicted by the AE based ROM. The second data point gave the highest mse when predicted by the POD based ROM.

6.1.2 Multigroup problem with cross-section perturbations

For this problem, a modified version of the one group CAE is used. The main difference is that most of the subsequent convolutional layers and subsequent transposed convolutional layers are replaced with residual blocks and transposed residual blocks respectively. Every residual block is two 3×3 convolutional layers deep, which results in residual blocks that are equal to the one shown in Figure 6. The new CAE also needs to accept seven different input maps and output seven different output maps, where every map is the value of an energy group. The parallel convolutional layers with an non-dilated and a dilated layer is extended with an extra dilated layer in parallel. The dilation rates are 17×17 and 31×31, resulting in receptive fields of size 35×35 and 63×63. The CAE has a latent space dimension of 7 while there are 42 FOM parameters. A schematic overview of the network can be seen in Figure 18.

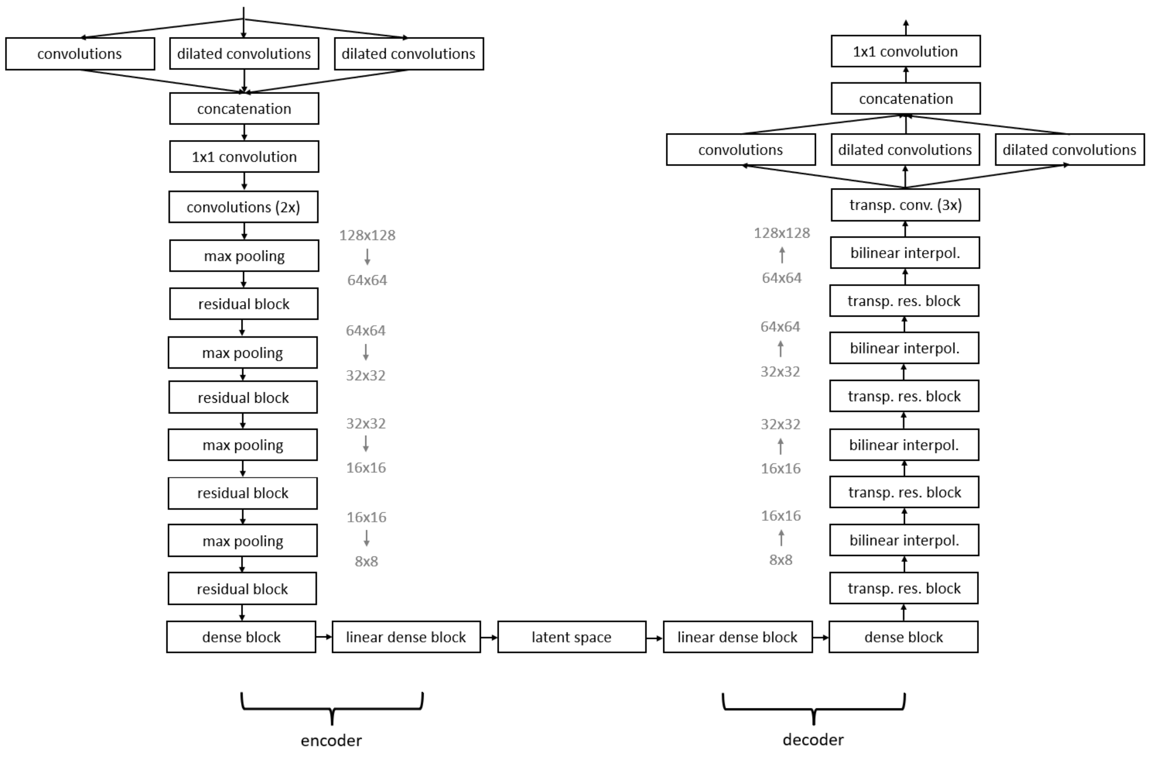


Figure 18: The used CAE network for the multigroup problem. If the kernel size of a convolutional layer is not mentioned specifically, a 3×3 kernel has been used. Every residual and transposed residual block (trans. res. block) is two 3×3 (transposed) convolutional layers deep. All nonlinear activations are 'elu' functions.

In short, the encoder is 3 convolutional layers, 4 residual block (of 2 convolutional layers) and 5 dense layers deep and maps to a latent dimension of 7. The decoder is 4 convolutional layers, 4 residual block (of 2 convolutional layers) and 5 dense layers deep, where 1 convolutional layer, namely the very last one, is used for feature map reduction. The model needs a total of 3.7 million weights and biases, where 1.7 million are used in the encoder and 2.0 million in the decoder.

The training data consisted of 4410 FOM solutions. During the training, the learning rate was reduced upon stagnation of the validation loss. Plots of a validation and a test result can be found in the Appendix, Figure A.2 and Figure A.3. Although the latent space looks smooth and has low polynomial shapes for single (e.g., Figure 19 and Figure 20) and dual perturbations (Appendix, Figure A.4), the multivariate polynomial regression model fails to approximate the latent variables from the FOM parameters. Namely, multivariate polynomial regression is required to find $C(q + c, c)$ different coefficients, where C is the binomial coefficient and and c the max degree of the polynomial. With 42 FOM parameters, the number of coefficients which need to be found quickly grows, making approximations of degree 4 and higher computationally infeasible. So, not the ROMs their performances, but instead, the performance of the CAE and the truncated POD projection with $p = 7 = d$ modes are given in Table 5.

	Training set	**Validation set**	**Test set**
CAE			
mae	0.69	0.70	0.68
mse	2.1	2.2	2.0
POD projection			
mae	0.98	0.99	0.96
mse	3.7	3.8	3.4

Table 5: Mean absolute and mean squared error of both the CAE prediction and the truncated POD projection for the multigroup problem with cross-section perturbations. All errors indicate the difference between the prediction and the ground truth values.

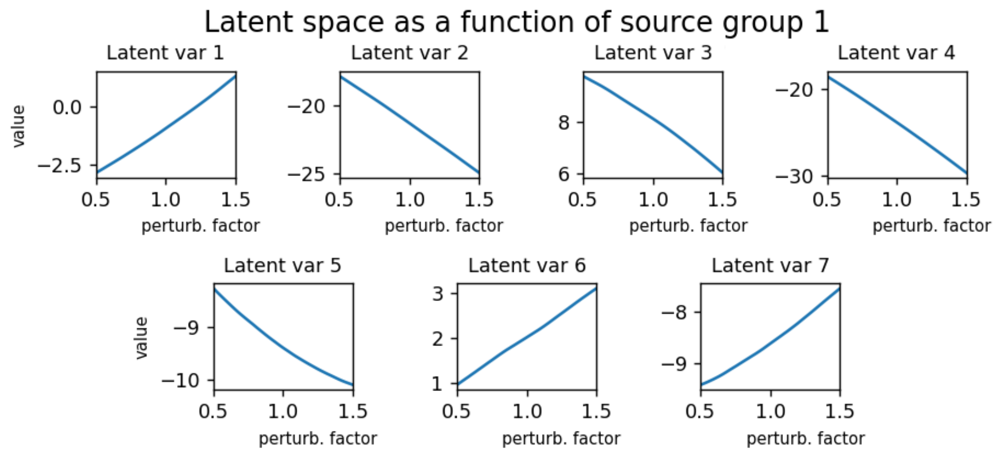


Figure 19: The influence of the perturbation of the source term of group 1 on the value of every latent variable. The source term is perturbed from 50% to 150% of its unperturbed value. All other FOM parameters are kept unperturbed.

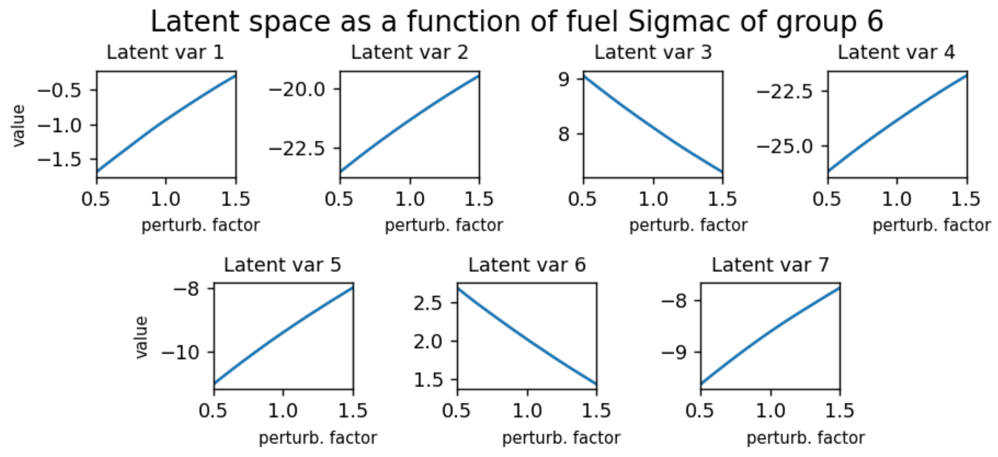


Figure 20: The influence of the perturbation of the macroscopic capture cross-section of group 6 of the fuel material on the value of every latent variable. The cross-section term is perturbed from 50% to 150% of its unperturbed value. All other FOM parameters are kept unperturbed.

6.1.3 One group problem with geometric perturbations

For this problem, a CAE is constructed with both residual blocks and parallel resBlocks. All the residual blocks have a depth of two 3×3 convolutional layers and the structure of the block is the same as shown in Figure 6, except that just for the addition of the skip connection, an extra 1×1 convolution layer is added to make sure that the output maps match. The used parallel resBlocks all have the exact same structure as the parallel resBlock shown in Figure 13. The latent space dimension is equal to the amount of possible perturbation, which is 18. That means that the dimension is reduced from $m = 16384$ to $d = 18$. The upsampling is done with strided transposed convolutional layers instead of the bilinear interpolation layers from the one and multigroup diffusion problems with cross-section perturbations. The transposed convolutional layer upsampling was found to give better results. A schematic overview the CAE network is given in Figure 21. The CAE consists of 10.3 million weights and biases, of which 4.7 million are in the encoder and 5.6 million in the decoder.

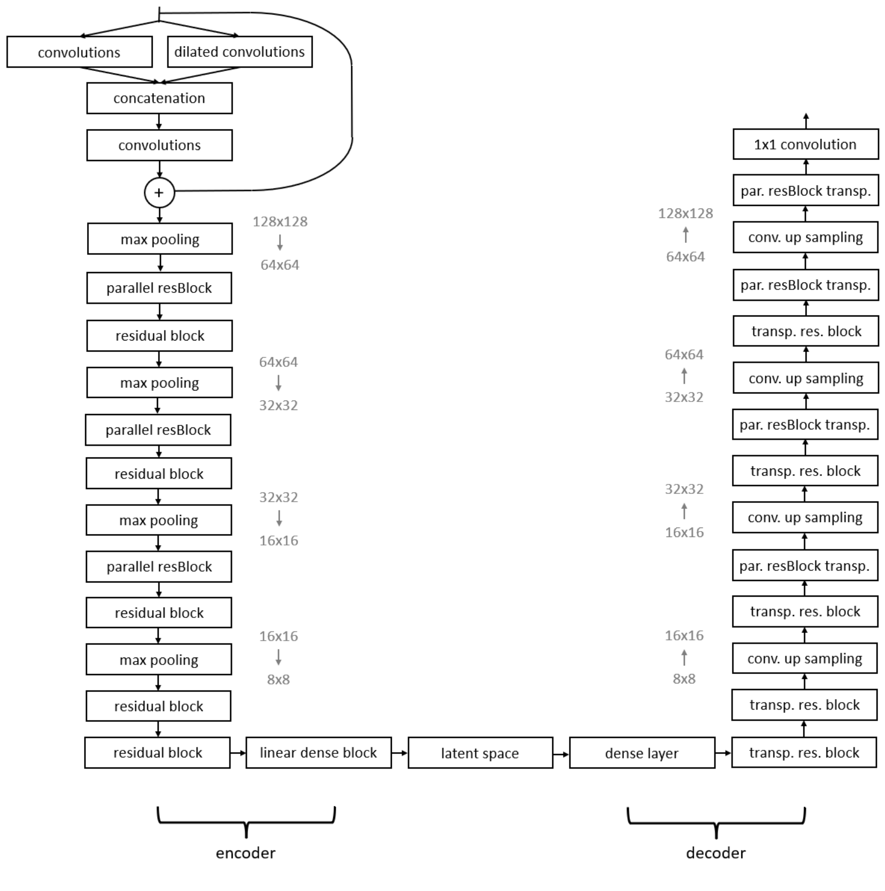


Figure 21: The used CAE network for the one group problem with geometric perturbations. If the kernel size of a convolutional layer is not mentioned specifically, a 3×3 kernel has been used. The parallel resBlocks have a depth of two 3×3 convolutional layers. The default residual blocks have the same depth. The linear dense block in the encoder consists of three linear dense layers all with an output dimension equal to the latent dimension. The last dense layer maps to the latent layer. No normalization layers are used. The nonlinear activations are all 'elu' functions. All dilated convolutions (including those in the parallel residual blocks) have a receptive field of approx. 1/3 of the dimension of their input feature map.

The training was performed on a training set with 5600 FOM solutions. The Adam optimizer was used to optimize the gradient descent on mini-batches of size 32. The global learning rate was reduced upon encountering a stagnation in the reduction of the validation loss. The polynomial regression model predicted the latent variables and POD coefficients with a very low error (mse $< 10^{-10}$) for a polynomial degree of 5. This can be seen in Table 6. The CAE based ROM was created and compared to a POD based ROM with the number of POD being equal to the latent dimension, i.e., $p = d = 18$. The mean squared and absolute error of both ROM models for the training, validation and test sets is shown in Table 7. The errors were the same for the CAE and the POD projection, for the FOM parameter regression had such a low error. Predictions of both ROMs on the training set and test set are shown in Figure 22 and Figure 24. The worst test set predictions are shown in Figure 24.

	Training set	**Validation set**	**Test set**
FOM parameters to latent vars			
mse	1.3×10^{-23}	3.3×10^{-11}	3.8×10^{-11}
FOM parameters to POD coef.			
mse $\times 10^{-28}$	1.7	2.1	2.1

Table 6: Mean squared error of the multivariate polynomial regression from the FOM parameters to the latent space and to the POD coefficients.

	Training set	**Validation set**	**Test set**
CAE based ROM			
mea $\times 10^{-2}$	4.4	4.5	4.5
mse $\times 10^{-2}$	0.73	0.77	0.79
POD based ROM			
mea $\times 10^{-2}$	6.5	6.5	6.6
mse $\times 10^{-2}$	1.9	1.9	2.0

Table 7: Mean absolute and mean squared error of both the CAE based ROM and the POD based ROM for the one group problem with geometric perturbations. The first one slightly outperforms the latter one. All errors indicate the difference between the prediction and the ground truth values.

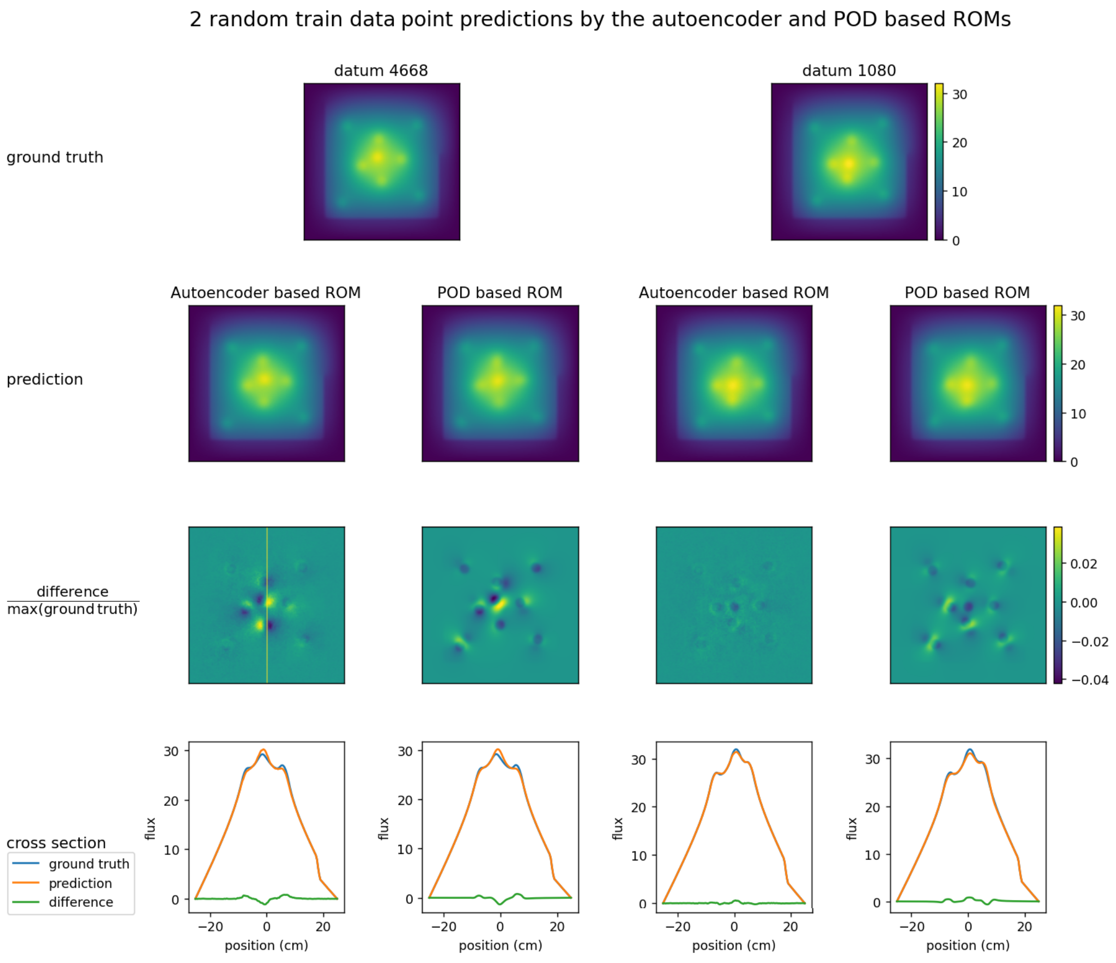


Figure 22: Two random data points from the train data set predicted by both the AE based ROM and the POD based ROM. Apart from the predictions, the difference divided by the max of the ground truth as well as an intersection is shown. The intersection is vertical and runs through the middle of the geometry (indicated with the orange line in the left difference plot. The AE based ROM returns better results than the POD based ROM.

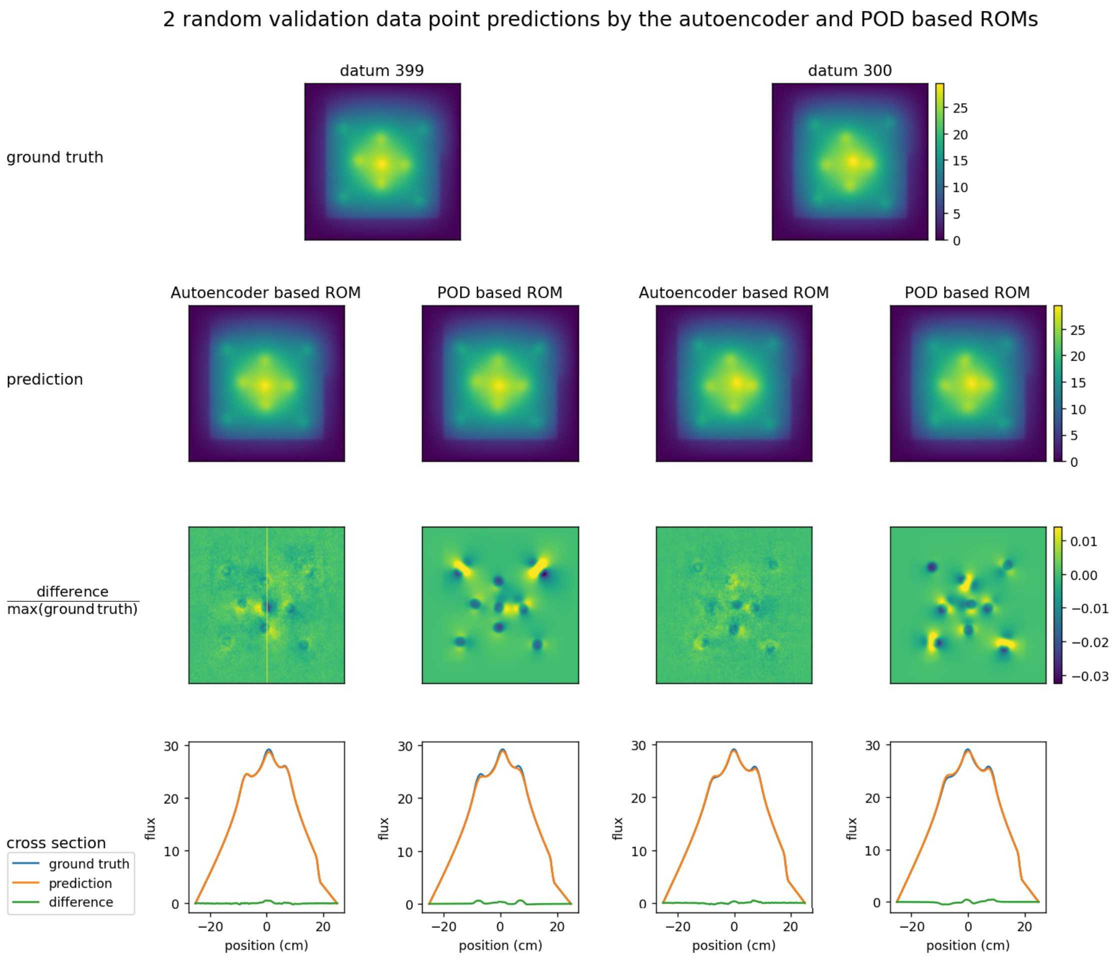


Figure 23: Similar plot to the one in Figure 22, except the two data points are drawn from the validation set. The performance of the ROMs is similar on the train and validation sets and this can be seen in the predictions as well.

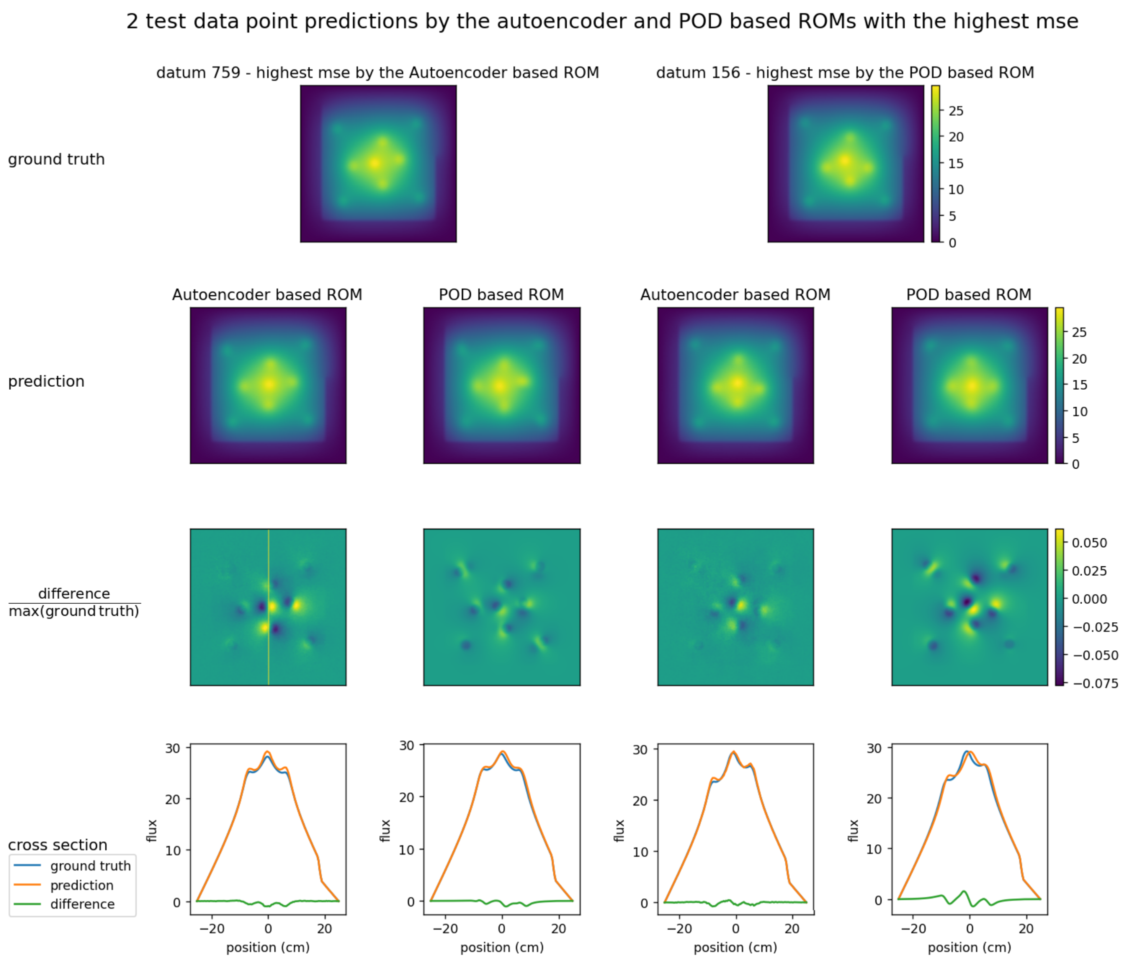


Figure 24: Subplots of the prediction, difference and vertical intersection through the geometry of two data points from the test set. The first data point gave the highest mse when predicted by the AE based ROM. The second data point gave the highest mse when predicted by the POD based ROM.

6.2 Other results

6.2.1 Parallel resBlocks, residual blocks and no skip-connections

Thus far, the performance of three different CAE networks has been given by either the CAE based ROM performance or the CAE performance. However, the three different networks cannot be directly compared since the they are all applied to different problems and are of different sizes. In this subsection, a fair comparison will be made between the three used architectures. The base for the comparison is the same model as used in the one group problem with geometric perturbations from Section 6.1.3. This is the architecture that includes parallel resBlocks and default residual blocks. The second architecture will not use parallel resBlocks, which can be achieved by converting every parallel resBlock into a residual block, where the number of filters of the second branch will be added to the filters of the first branch. This way, the residual blocks network will have the same amount of filters and convolutional layers. The third architecture will not include skip connections and hence, all residual blocks are converted to a sequence of consecutive convolutional layers with a depth equal to the depth of the residual block. The amount of model parameters is approximately the same for all three architectures and not perfectly equal due to the removal of some 1×1 convolutional layers. All three models are trained on the same data. The learning curves are plotted in Figure 25.

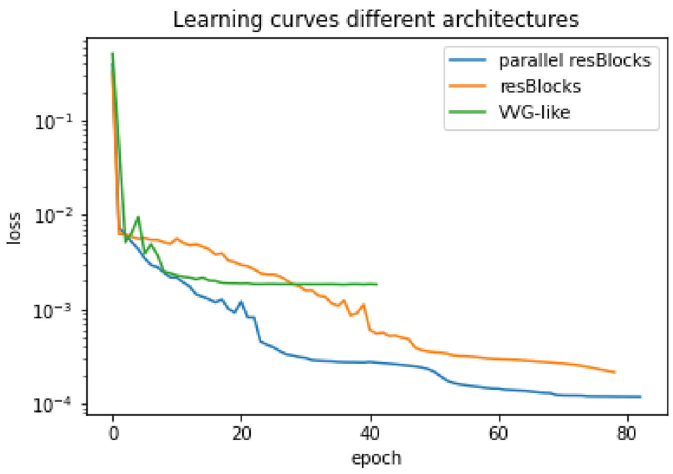


Figure 25: The learning curves for three different architectures with the same depth and approximately the same number of model parameters.

6.2.2 Regularization

Since the FOM data is obtained by numerical solutions of a PDE and not by noisy measurements of a physical unit, it is safe to say that there is no measurement noise in the data and that over-fitting of this noise is very unlikely. This is confirmed in practice, where all validation losses seems to follow the loss very closely. Using regularization seems unnecessary, or even counterproductive because every gradient descent step will take a little longer since the loss function, and thus its derivative, will need more calculations to be determined. Still, there are arguments to be made for the use of regularization. It may slow down every gradient descent step, but it may also need less steps to converge. Also, regularization of the latent activations, i.e., the regularization of value of the latent variables, forces the latent variables to stay close to zero, which may result in better polynomial regression from the FOM parameters to the latent variables. The first argument was checked by plotting the learning curve of a model with different L2 regularization imposed on its latent activations. The used model is a similar variant of the one showed in Figure 21, with the main difference that all parallel resBlocks are removed. The networks needed to fit the data from the one group problem with geometric perturbations. The learning curves of the loss is plotted in Figure 26.

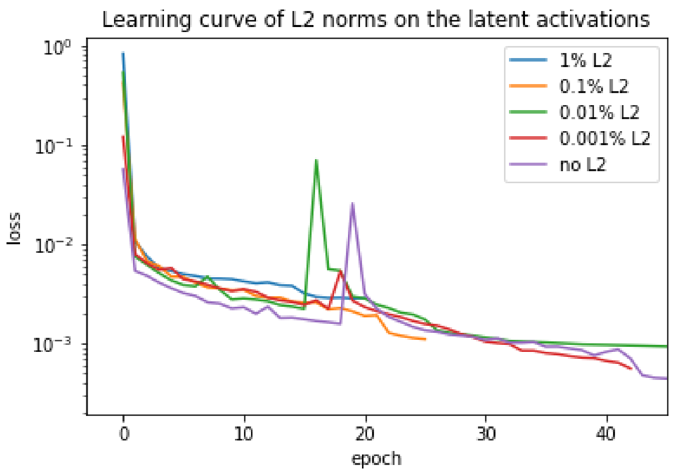


Figure 26: The learning progression for different L2 penalties on the latent activations. All training sessions used the same batch size. The peaks in the 0.01%, 0.001% and the 0% L2 norm around epoch 17-20 are due to the continuation of a training. The training stopped prematurely (due to a server disconnection) and the training needed to be continued from where it stopped. The restart was made with the learning rate at which the first training stopped. However, the Adam optimizer needed to relearn its parameters again, which caused the peaks. After a few epochs, the learning is back on the level it stopped at.

6.2.3 Data sparsity

A goal of reduced order modelling is the reduction in computation time and the ability to not be required to calculate the FOM model. However, many FOM models need to be calculated during the data acquisition stage of the CAE based ROM creation. Since FOM calculations are usually expensive, one wants to acquire only the bare minimum of FOM solutions needed for the CAE training. Although this is not the main goal of this research, it is important to know if the used data sets are big enough. Therefor, one small experiment has been done. A simple network with no skip-connections was trained on all 6300 samples in the training set from the one group problem with cross-section perturbations. The network was trained again, but only on the 128 sample of the training set whose FOM parameters are furthest apart from each other in FOM parameter space. 128 samples times 16384 points is approximately 2-3 times the number of model parameters. During (unregularized) training, the training and validation loss converted similar to the losses of the full dataset training, although the training loss of the 128 data point training session started to overfit in the very end, when its validation and training loss were at the same level as the 6300 data point training. This can been seen in Figure 27.

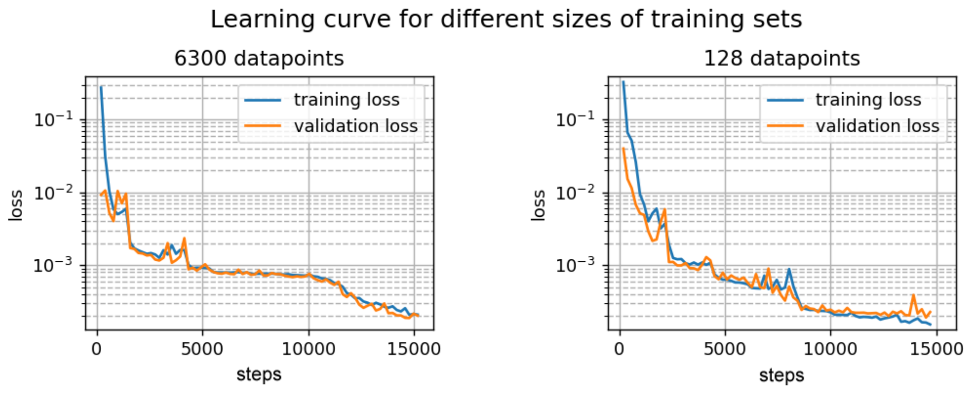


Figure 27: The learning progression of two separate training sessions of the same model, but with different dataset sizes. The 128 points from the smallest set are taken the big set and are the 128 points which were the furthest apart in FOM parameter space. The training sessions used the same validation set (1350 data points). The learning rate was reduced upon stagnation in the learning. The learning rate of the 128 data point training was reduced much earlier, resulting a faster reduction of the loss. Both training sessions stopped learning at approximately the same validation loss, while the training loss of the 128 data point training session started to slightly overfit.

6.2.4 Batch normalization, layer normalization and no normalization

None of the models mentioned thus far have used a form of normalization inside the network itself. Only the input data has been normalized (or rather, standardized). In the Theory section, batch normalization and layer normalization have been mentioned to improve the training progress and outcome. However, batch normalization layers perform better for bigger mini-batch sizes and layer normalization layers are not recommended for convolutional layers. This is confirmed by the training of three different residual blocks model for the multigroup problem. One network uses batch normalization layers after every convolutional layer, one network uses layer normalization layers after every convolutional layer, and one model does not use any normalization layer. All networks are equal apart from the normalization layers. The learning curve is shown in Figure 28. The training session were performed with a mini-batch size of 16. It is recommended to train a similar model with a group normalization layer [35] in order to see how those layers influence the performance.

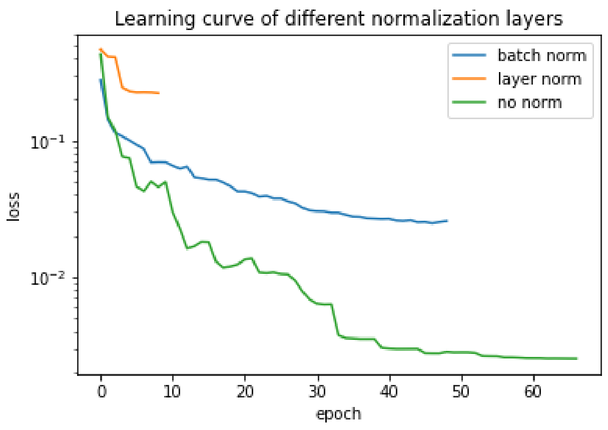


Figure 28: The learning progression for different normalization layers. The mini-batch was of size 16.

6.2.5 Upsampling layers

The CAE for the one group problem with cross-section perturbations and the multigroup problem both used bilinear interpolation layers in their decoders in order to increase the dimensionality of the feature maps. The CAE for the one group problem with geometric perturbations used transposed convolutions. This choice seems more or less arbitrary, which can be backed up by Figure 29.

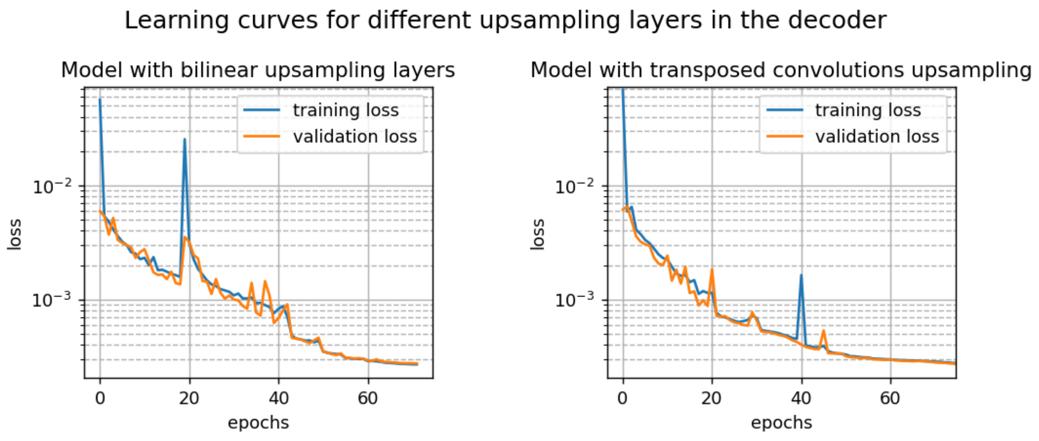


Figure 29: The learning curves of the training loss for two models which are equal in everything, except for their upsampling layers used in their decoders. The first model uses bilinear interpolation layers while the second model used transposed convolutional layers. Both models arrive at a similar lowest training loss and validation loss. The peak around epoch 20 in the curve of the bilinear model and the peak around epoch 40 in the transposed convolutional model are due to a restart in the training which was needed for the training stopped prematurely (reached temporarily user-limits in Google Collab). The restart was performed with a copy of the settings from the epoch in which the training halted, except for learn-able parameters of the Adam optimizer. Those had to be relearned.

6.2.6 Custom activation functions

Since all used models had no normalization layers, other ways of increasing the training efficiency were looked for. Two custom activation functions have been tried on the same model to see if they would enhance the training. The first one being a variant on the previous often used hyperbolic tangent functions. The disadvantage of the hyperbolic tangent is the saturation of the gradient when moving to plus or minus infinity. In practice, the gradient is already 1e-4 before 5. A modified version of the hyperbolic tangent was tried, were the saturated regimes of the hyperbolic tangent function are replaced with lines. That is,

$$g(z) = \begin{cases} 1/e\left(z - \cosh^{-1}(\sqrt{e})\right) + \tanh(\cosh^{-1}(\sqrt{e})), & z > \cosh^{-1}(\sqrt{e}) \\ \tanh(z), & \cosh^{-1}(\sqrt{e}) \geq z \geq -\cosh^{-1}(\sqrt{e}) \\ 1/e\left(z + \cosh^{-1}(\sqrt{e})\right) + \tanh(-\cosh^{-1}(\sqrt{e})), & z < -\cosh^{-1}(\sqrt{e}). \end{cases} \tag{31}$$

The second custom activation function is of similar fashion, but does not require any evaluation of the hyperbolic tangent anywhere:

$$g(z) = \begin{cases} \frac{1}{4}(z-1)+1, & z > 1 \\ z & 1 \geq z \geq -1 \\ \frac{1}{4}(z+1)-1, & z < -1. \end{cases} \tag{32}$$

Both Equation (31) and Equation (32) are nonlinear and are symmetric around the z-axis. This way, if the input data is zero-centered and the weights are randomly initiated around zero, the covariate shift will be minimal. The other advantageous property which both function have, is the fact that they cannot return near-zero gradients. This will prevent fading gradients. However, they did not outperform the 'elu' activations when tried as alternative activation function for a network. They performed worse when trained on the same network as the 'elu', as can be seen in Figure 30. In the figure, Equation (31) will be referred to as the 'linear modified hyperbolic tangent', or 'lin. mod. tanh', and Equation (32) will be referred to as the 'double linear nod'. The network was a VGG-like network for the one group problem with cross-section perturbations. Besides both custom activations not resulting in faster learning nor a lower final loss, they also were found to be less stable, for the initial learning rate had to be set lower to prevent exploding gradients. It should be noted that there was no regularization, which may could have prevented this.

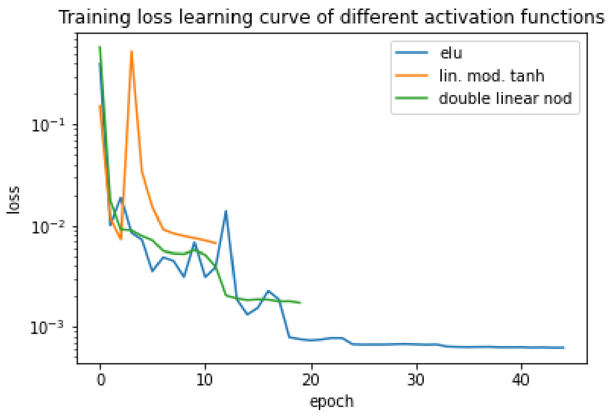


Figure 30: The training loss for the same network, but for different activation functions. No normalization layers are present within the network. The two custom activation functions do not outperform the 'elu' activation.

7 Discussion

The CAE based ROM had a hard time outperforming the POD based ROM. It only outperformed the POD based ROM for the one group problem with geometric perturbations. This may partly be due to the simpler models being used for the one group problem with cross-section perturbations and the multigroup problem with cross-section perturbations.

The main focus of researching the feasibility of the CAE based ROMs was on outperforming the POD based ROMs in terms of obtaining a lower error, i.e., have a better reconstruction of the FOM solution. However, other important properties of a ROM are the computational costs and time costs, and those should be researched as well. A part of researching and optimizing the computational costs should be focused on data sparsity, for one does not want to acquire a minimum amount of FOM solutions.

Only a multivariate polynomial regression model has been used for this regression problem. Other forms could be tried. During the thesis, a simple neural network has been tried for this goal as well, but only very experimentally. It however has potential, which should be checked. Also, imposing extra restrictions on the the latent variables during the training could benefit the FOM parameter to latent variables model.

It would be interesting to apply the best models to problems which are harder to capture with a truncated POD basis. Advection dominated problems have shown to create a clear distinction between nonlinear and linear subspace ROMs [6], [2], [5]. Applying the proposed method to such problems will help to gain more insight into its performance. Extension to time dependency is also interesting, since the effect of nonlinearities can increase in time which could be a better way of pointing out the difference in performance between CAE based reduced order modelling and POD based reduced order modelling. The main idea is that a CAE better approximates the nonlinear manifold on which a solution lays than the POD. The discarded POD modes may contain valuable information which is lost.

8 Conclusion

It is shown to be feasible to create a CAE based ROM which outperforms its POD based counterpart, at least in terms of predictions. The proposed CAE based ROM was applied to three different steady state neutron diffusion problems. For every problem, a different CAE structure was used. The one group problem with cross-section perturbations was approached with a VGG-like network. The CAE constructed for the multigroup problem with cross-section perturbations utilized residual blocks. The one group problem with geometric perturbations its CAE used both residual blocks and parallel residual blocks. The ROM based on this CAE managed to outperform its POD based counterpart by predicting the data points from the training, the validation and test set with a mean squared error which was approximately 2.5x smaller than the POD based ROM, while the mean absolute error was about 1.4x times smaller than the POD based ROM. It was also numerically shown that for the same one group problem with geometric perturbations a model with a combination of parallel residual blocks and residual blocks would outperform a model with only residual blocks only which in turn would outperform a VGG-like model. All three models had approximately the same number of weights.

Research is recommended on the computational and time cost part of the ROMs as well as extension of the CAE based ROM to time dependent problems. A small experiment was performed where a model was trained on only a small number of data points, namely, the input dimensions times the number of data points was only 2-3 times the number of model parameters inside the CAE. The training resulted in similar loss compared to a training on 49 times as many data points. This proves hopeful for future research on the data sparsity. In order to gain further insight in the feasibility of the proposed ROM method, it is recommended to apply the ROM to problems which are harder to be captured by a POD approach. Also, extending the proposed method to time dependent models is something which could be done in the future, by either intrusive or non-intrusive use of the latent variables.

References

[1] Y. Kim, Y. Choi, D. Widemann, and T. Zohdi, “A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder,” 09 2020.

[2] S. Dutta, P. Rivera-Casillas, O. M. Cecil, M. W. Farthing, E. Perracchione, and M. Putti, “Data-driven reduced order modeling of environmental hydrodynamics using deep autoencoders and neural odes,” 2021.

[3] T. O’Leary-Roseberry, U. Villa, P. Chen, and O. Ghattas, “Derivative-informed projected neural networks for high-dimensional parametric maps governed by pdes,” 2021.

[4] B. Lusch, J. N. Kutz, and S. L. Brunton, “Deep learning for universal linear embeddings of nonlinear dynamics,” *Nature Communications*, vol. 9, Nov 2018.

[5] R. Maulik, B. Lusch, and P. Balaprakash, “Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders.” preprint, 2020.

[6] K. Lee and K. Carlberg, “Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders,” 02 2019.

[7] P. Wu, J. Sun, X. Chang, W. Zhang, R. Arcucci, Y. Guo, and C. C. Pain, “Data-driven reduced order model with temporal convolutional neural network,” *Elsevier*, 2019.

[8] K. Lee and K. T. Carlberg, “Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders,” *Journal of Computational Physics*, vol. 404, 2020. Read the preprint version from 2019.

[9] Y. Choi, T. Zohdi, Y. Kim, and D. Widemann, “Efficient nonlinear manifold reduced order model.” preprint, 2020.

[10] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. USA: Cambridge University Press, 1st ed., 2019.

[11] R. Pinnau, *Model Reduction via Proper Orthogonal Decomposition*, pp. 95–109. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

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

[13] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” 2016.

[14] M. Lin, Q. Chen, and S. Yan, “Network in network,” 2014.

[15] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015.

[16] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016.

[17] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[18] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.

[19] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, (Madison, WI, USA), p. 807–814, Omnipress, 2010.

[20] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” 2013.

[21] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," 2016.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[25] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," 2018.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," 2016.

[27] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[28] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," 2016.

[29] L. Jing, J. Zbontar, and Y. LeCun, "Implicit rank-minimizing autoencoder," 2020.

[30] J. J. Duderstadt and L. J. Hamilton, *Nuclear Reactor Analysis.* John Wiley & Sons, 1976.

[31] The Mathworks, Inc., Natick, Massachusetts, *MATLAB.*

[32] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace, 2009.

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[34] F. Zhang, F. Yang, C. Li, and G. Yuan, "Cmnet: A connect-and-merge convolutional neural network for fast vehicle detection in urban traffic surveillance," *IEEE Access*, vol. PP, pp. 1–1, 05 2019.

[35] Y. Wu and K. He, "Group normalization," 2018.

9 Appendix A

9.1 Extra figure one group problem with cross-section perturbations

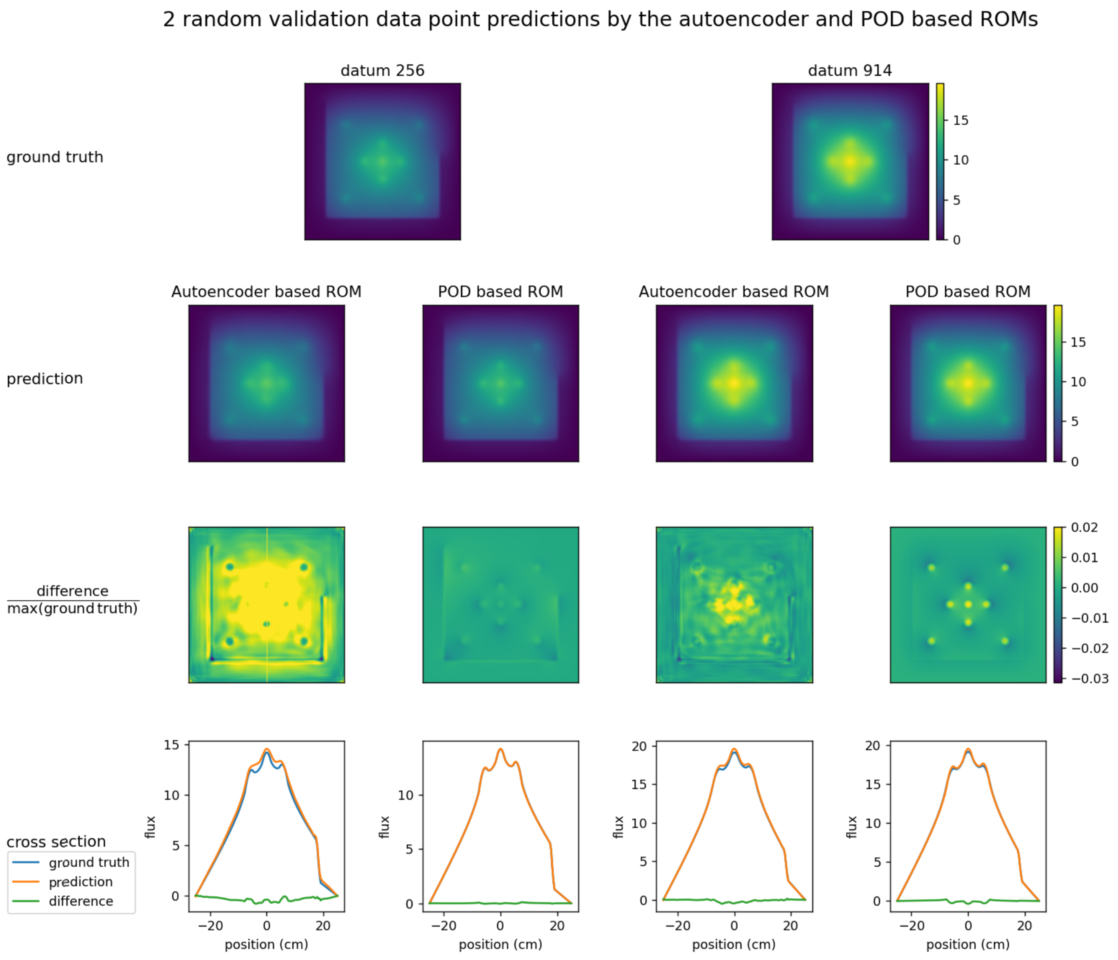


Figure A.1: Subplots of the prediction, difference and vertical intersection through the geometry of two data points from the validation data set of the one group problem with cross-section perturbations.

9.2 Extra figures multigroup problem with cross-section perturbations

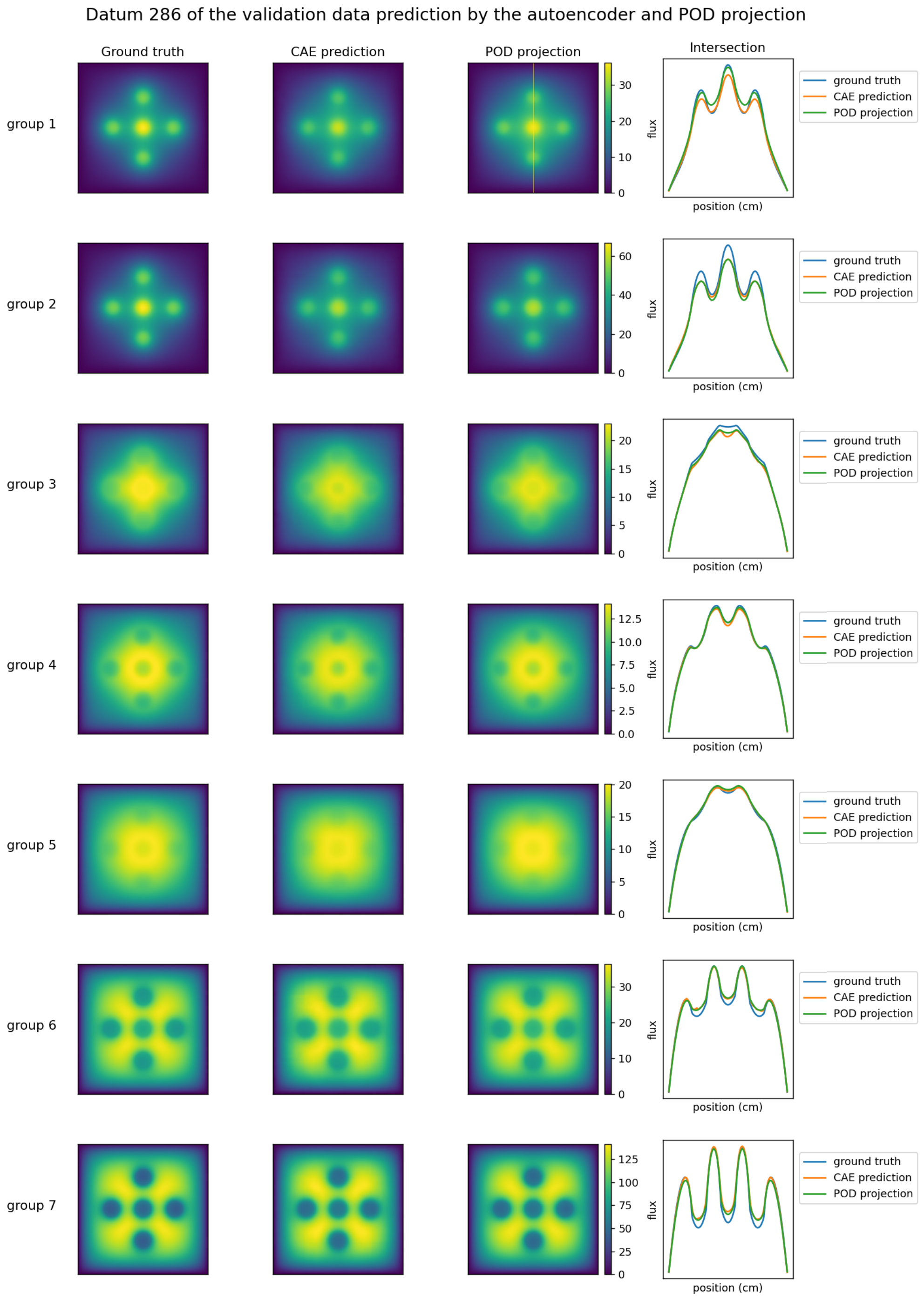


Figure A.2: A random datum from the validation data set of the multigroup problem. The ground truth, the CAE prediction, the POD projection and the intersection from top to bottom through the middle is given for every energy group. Both the CAE and the POD are able to capture the shape of the ground truth, except for the perfect amplitudes.

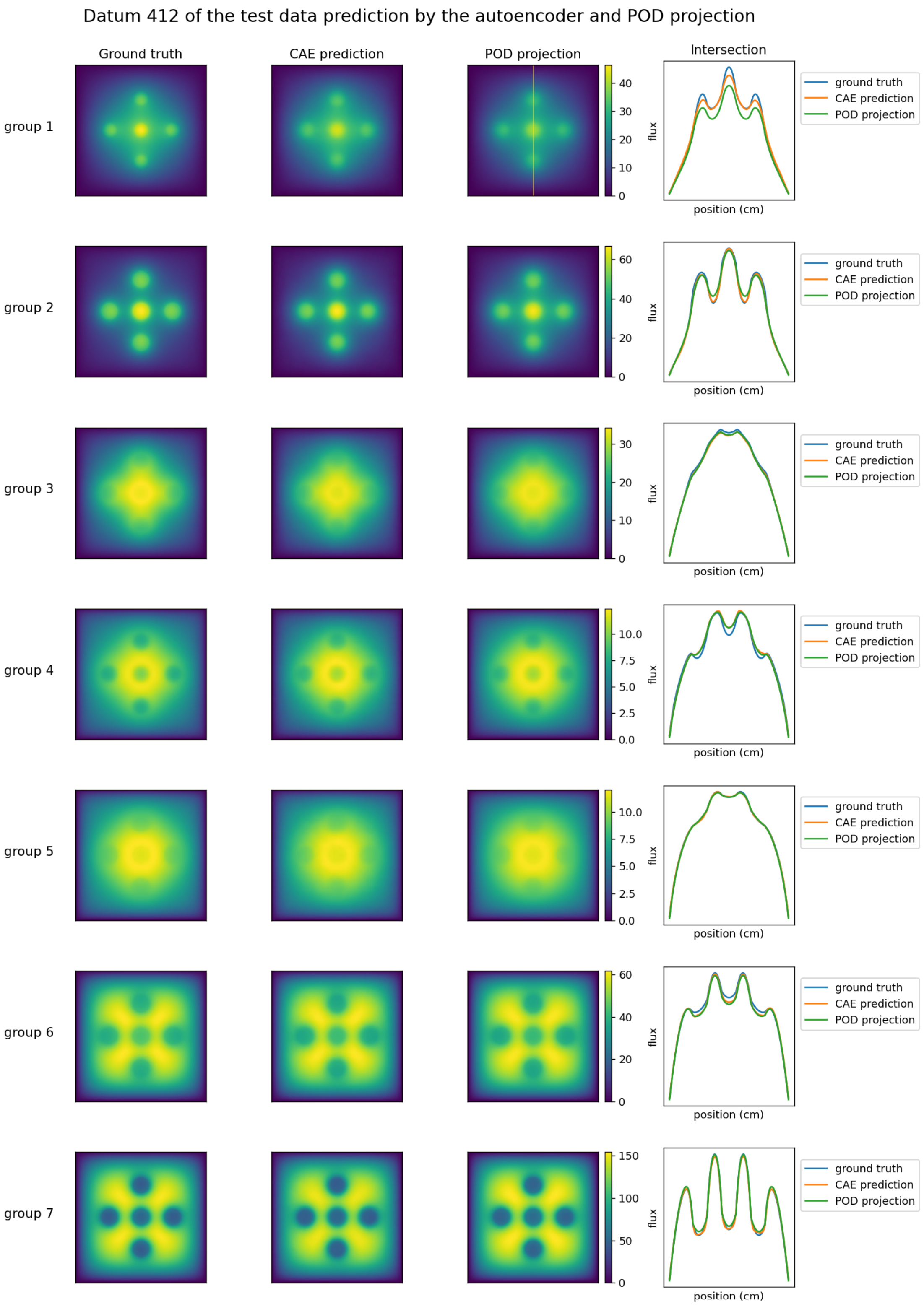


Figure A.3: A random datum from the test data set of the multigroup problem. The ground truth, the CAE prediction, the POD projection and the intersection from top to bottom through the middle is given for every energy group.

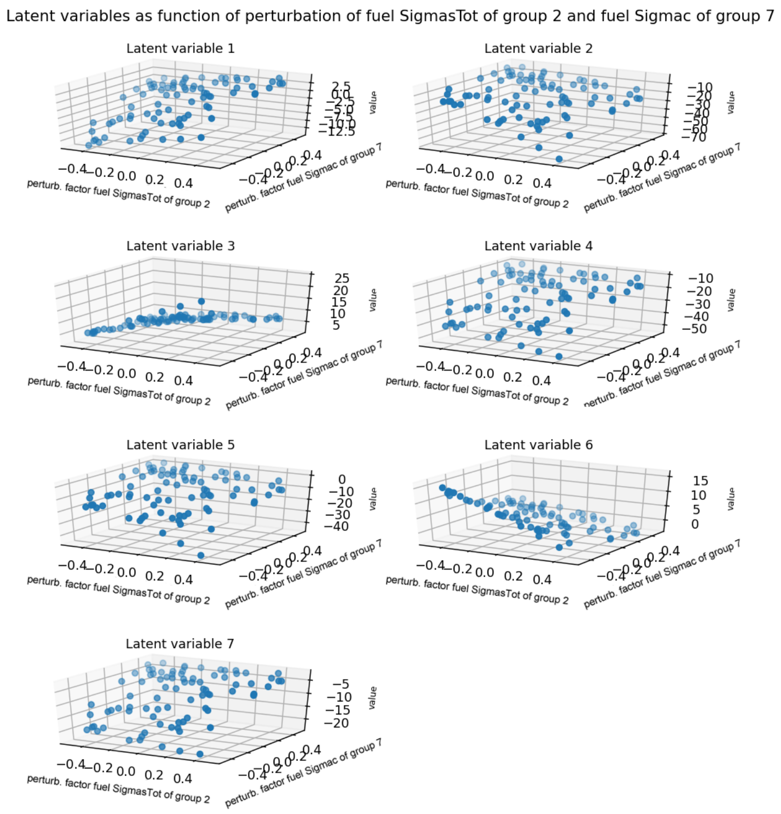


Figure A.4: The value of every latent space variable of the CAE for the multigroup problem as a function of the perturbation of the total scatter cross-section of group 2 and the capture cross-section of group 7 of the fuel. All other FOM parameters are unperturbed. The resulting surfaces are smooth, similar to the single FOM parameter perturbations.