Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Liv Breivik

Combining Artificial Neural Networks with Reduced Order Models with Applications to Classification Problems

Master's thesis in MTFYMA
Supervisor: Brynjulf Owren
July 2023

NTNU
Norwegian University of
Science and Technology

Liv Breivik

Combining Artificial Neural Networks with Reduced Order Models with Applications to Classification Problems

Master's thesis in MTFYMA
Supervisor: Brynjulf Owren
July 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Abstract

This thesis serves as a proof of concept for the neural network reduction techniques provided in [36] and [13]. The model that has been reduced is the VGG-16 model, trained on augmented versions of both the CIFAR-10 dataset and the SVHN dataset. The procedure for constructing a reduced neural network is to take the output from a layer in a full neural network, project it to a smaller dimension using a projection matrix, and mapping the output onto the output-space of the full neural network. The reduced neural network can then be re-trained using knowledge distillation methods, if the accuracy is inadequate. The projection matrix construction methods used were Proper Orthogonal Decomposition and Active Subspaces. The input-output mapping methods that have been implemented are Feedforward Neural Networks and Polynomial Chaos Expansion. All combinations between the projection matrices and input-output mappings have been implemented. This thesis provides incremental results during the construction of the reduced neural network to evaluate the different steps of the process. Quantitative and quantitative results are provided to see the effect of varying independent variables within the reduced neural network methods. After presenting and discussing the results as well as choices taken during the implementation, we give a recommendation for the procedure of constructing reduced neural networks.

Sammendrag

Denne avhandlingen er en konseptutføring for reduksjonsteknikkene for nevrale nettverk som beskrevet i [36] og [13]. Modellen som er blitt redusert er VGG-16-modellen, som er trent på augmenterte versjoner av både CIFAR-10-datasettet og SVHN-datasettet. Prosedyren for å konstruere et redusert nevralt nettverk innebærer å ta ut-dataene fra et lag i et fullstendig nevralt nettverk, projisere dem til en mindre dimensjon ved hjelp av en projeksjonsmatrise, og kartlegge ut-dataene til utdatarommet til det fullstendige nevrale nettverket. Det reduserte nevrale nettverket kan deretter trenes på nytt ved hjelp av metoder som kunnskapsdestillasjon hvis nøyaktigheten ikke er tilstrekkelig. Metodene til konstruksjon av projeksjonsmatrisene var Proper Orthogonal Decomposition og Active Subspaces. De implementerte metodene for kartlegging av in-data og ut-data er Feedforward Neural Networks og Polynomial Chaos Expansion. Alle kombinasjoner mellom projeksjonsmatrisene og inndata-utdata-kartleggingene er implementert. Denne avhandlingen gir gradvise resultater under konstruksjonen av det reduserte nevrale nettverket for å evaluere de ulike stegene i prosessen. Kvantitative og kvalitative resultater blir presentert for å se effekten av variasjoner i uavhengige variabler innenfor metodene for reduserte nevrale nettverk. Etter å ha presentert og diskutert resultatene, samt valgene som ble tatt under implementeringen, gir vi en anbefaling for prosedyren for å konstruere reduserte nevrale nettverk.

Acknowledgements

I would like to thank my supervisor, Brynjulf Owren, for great supervision throughout the thesis.

I would also like to thank Per Kristian Hove for helping me with computational problems.

I would also like to thank the roommates and friends with whom I shared this chapter of my life. Trondheim would not have been the same without you.

Lastly, I would like to thank my family for supporting my academic decisions.

Table of Contents

List of Figures

List of Tables

1 Introduction

An introduction to artificial neural networks is often done by comparing the processes of the neural network to processes in the brain, as proposed in [35]. The father of AI is generally accredited to be Alan Turing, the inventor of the Turing test and the author of the 1948 paper *Intelligent Machinery* [56]. More than 75 years later, the ideas behind artificial intelligence and machine learning affect the daily discourse. Image recognition models are utilized to evolve self-driving cars, based on the driving recorded by Tesla users' autopilot function[49]. Radiologists are using AI to enhance their practice[1]. Search engines such as Google are using machine learning and data from their users to personalize advertisements displayed[16]. Natural language processing models such as ChatGPT [40] have become household names and have sparked societal debates regarding the future of AI[31]. All of this is made possible through the advancement of computers, and the following digitalization.

Python libraries such as PyTorch [42] are available, giving people the opportunity to construct both complex and simple neural networks from their home office. Famous models can be directly accessed through the sublibrary Torchvision. An increased knowledge of coding in the human population as well as an increased interest in neural networks has lead to amateur machine learning competitions as well as large-scale neural network competitions, giving us new and improved models. Image classification, for example, has evolved some of its more famous models through the ImageNet competition[46].

With the tasks of the neural networks becoming more specialized and complex, the repertoire of deep and complex models increases as well, making their computations more computationally expensive[55]. The digitalization leading to a higher amount of data yields a higher accuracy, yet also larger computation time for neural networks. This also restrains the artificial neural networks; a desired accuracy of a model requires a certain amount of data, available space on the computer, and time to train the neural network.

For decades, mathematicians have been using model order reduction methods to simplify and visualize high-dimensional data[33], as well as reduce computational complexity of mathematical models. Some of the more renowned methods include Finite Elements Method[10] and Principal Component Analysis[17].

The scope of this thesis will revolve around reduced order modeling of neural networks, as proposed in [36, 13]. The reduced system we are constructing consists of three parts: a split of the system into a pre- and post-model, a model reduction layer for projecting the output of the pre-model onto a lower dimension, and an input-output mapping to map the reduced output onto the space of the output from the full original model. The numerical methods used for the reduction layer are Active Subspaces (AS) and Proper Orthogonal Decomposition (POD). Both methods base themselves on creating a projection matrix using spectral decomposition. The input-output mappings used are a Feedforward Neural Network (FNN) and Polynomial Chaos Expansion (PCE). Additionally, a technique called Knowledge Distillation is employed for re-training the reduced neural network after construction, improving accuracy.

Chapter 2 thoroughly introduces the concept of artificial neural networks. It focuses on mathematical properties, while visualizing various concepts important to the subject matter. Chapter 3 consists of mathematical preliminaries. These are important to have a grasp of the model order reduction. Some of the topics touched upon here are least squares method, singular value decomposition, and frequent directions method. Chapter 4 introduces model order reduction of neural networks. Section 5 consists of a description of the implemented code (for reproduction purposes), the datasets, the model used, and specifications to the numerical methods. Section 6 consists of results for the full models, and the reduced models before and after the re-training step. Both a quantitative and a qualitative analysis of the reduced model is portrayed. Section 7 contains a discussion of the results. Some of the topics touched upon are the possibilities for a generalized reduced model, and the extension to datasets of larger dimensions.

2 Artificial Neural Networks

The aim of an Artificial Neural Network (ANN) is to approximate an unknown function $\mathbf{F}(x)$, given the input x and output y.

The approximation of the unknown function $\mathbf{F}(x)$ is done through the construction of different *layers*, where each layer has a set amount of *neurons*. The different layers also have *weights* and *biases* that help determine the worth of each input inside the neuron. The weights and biases have been determined through a *training period* characterized by gradient-based methods. Mathematically, the different layers represent parameterized functions $F_i(x;\theta_i)$, and so the approximation $\mathbf{F}(x)$ is in fact the composition of these.

By propagating the inputs x through the layers in the neural network, one gains an approximated output $\hat{y}$.

Most of the information attained in this section is from [19], [21] and [36].

2.1 The Neuron

The *neurons* of artificial neural networks are essentially nodes with numerical values dependent on the input data. In order to explain them more fully, we need to define some necessary terms:

- Input vector $x = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^n$. The data that is sent into a layer.
- Weights w_{ij}, belonging to each respective input i at every neuron j. This means that if a node has k inputs, there will equivalently be k associated weights that belong to every connection between the neuron j and the neurons $i = [0, \ldots, k]$.
- Bias b_j, a scalar value associated to every neuron j. This is independent of the various inputs.
- Activation function $\sigma(x)$. The purpose of this is to introduce non-linearity to the output of the artificial neural network.
- Predicted output $\hat{y}$. This denotes the output one gets from the neuron.

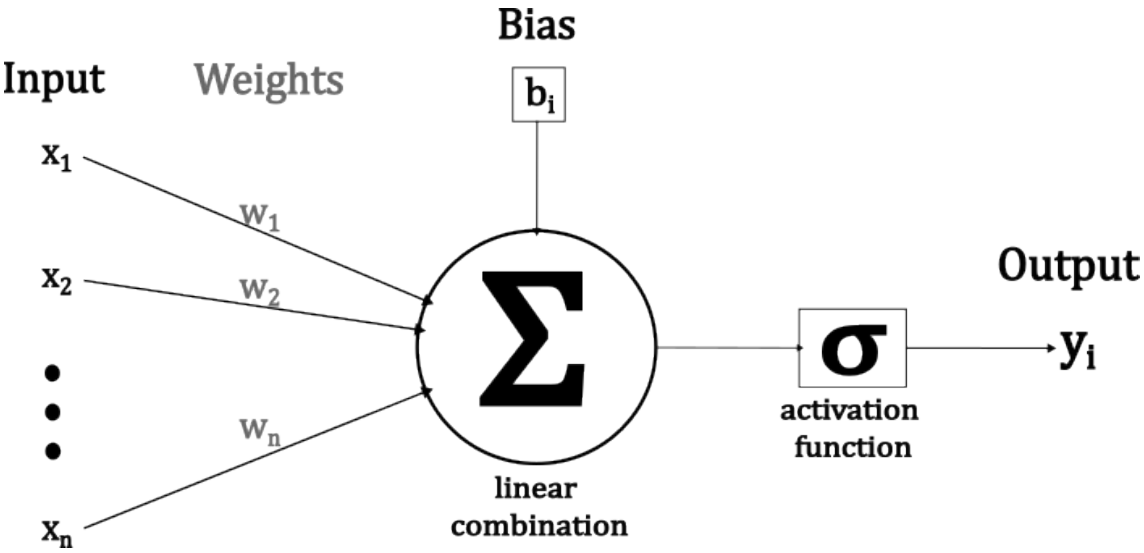


Figure 2.1: The setup of the individual neurons in a neural network.

The i'th neuron in an ANN is visualized in figure 2.1. Here one can see the inputs $x_1, \ldots, x_n$ being sent into the neuron, with their associated weights w_i. The aim of the weights is to measure the importance of different neuron connections. The bias term introduces flexibility by allowing the

neuron to adjust its own output independently, and ensures that the inputs in the neuron do not necessarily lead to zero.

All the inputs, weights, and biases fed into the neuron give us the linear combination

$$z = \sum_{j=1}^{n} w_j \cdot x_j + b. \tag{2.1}$$

The linear combination can better be expressed in matrix form. Denoting an array of inputs as $x = [x_1, \ldots, x_n]$ and the associated weight vector as $W = [w_1, \ldots, w_n]$. Still denoting our bias as the scalar b, we set up the vectorial form

$$z = x^T W + b. \tag{2.2}$$

The linear combination from the neuron is propagated further to an activation function $\sigma(x)$. The purpose of this is to introduce non-linearity, but also to determine whether the neuron in question should activate more neurons or be deactivated, based on its inputs. For example, if the activation function $\sigma(x)$ gives 0 as an output from the linear combination z, all the inputs and weights that have been assembled in z will have no further importance.

We thereby denote the output from every neuron as

$$a = \sigma(x^T W + b) = \sigma(z), \tag{2.3}$$

otherwise known as the *activation* of a neuron.

From equations (2.2)-(2.3) we see that sending an input x through a neuron will provide some sort of transformation. Constructing functions for each separate transformation in a neuron, we have:

- f_{linear}, denoting the linear combination $x^T W + b$.
- σ, the activation function.
- f_{out}, which sends out the output from the neuron to its next connections.

The combination of the three separate transformations gives us a general expression for the neuron as a function,

$$f = f_{\text{out}} \circ \sigma \circ f_{\text{linear}}. \tag{2.4}$$

This applies to any neuron in a neural network.

2.2 Topology of artificial neural networks

Artificial neural networks consist of several layers with different amounts of neurons in each layer. These are all interconnected.

The topology of an ANN is comprised of three layer types:

- **Input layer**: The first layer of the ANN, where the input data $x^{(0)}$ is received.
- **Hidden layers**: The layers within the ANN that are not directly connected to the input nor the output. The amount of hidden layers and nodes within the hidden layers is dependent on the network architecture and complexity.
- **Output layer**: The final layer of the ANN, producing the network output. The amount of nodes are dependent on the task of the neural network, but are generally equivalent to the amount of classes that the problem deals with.

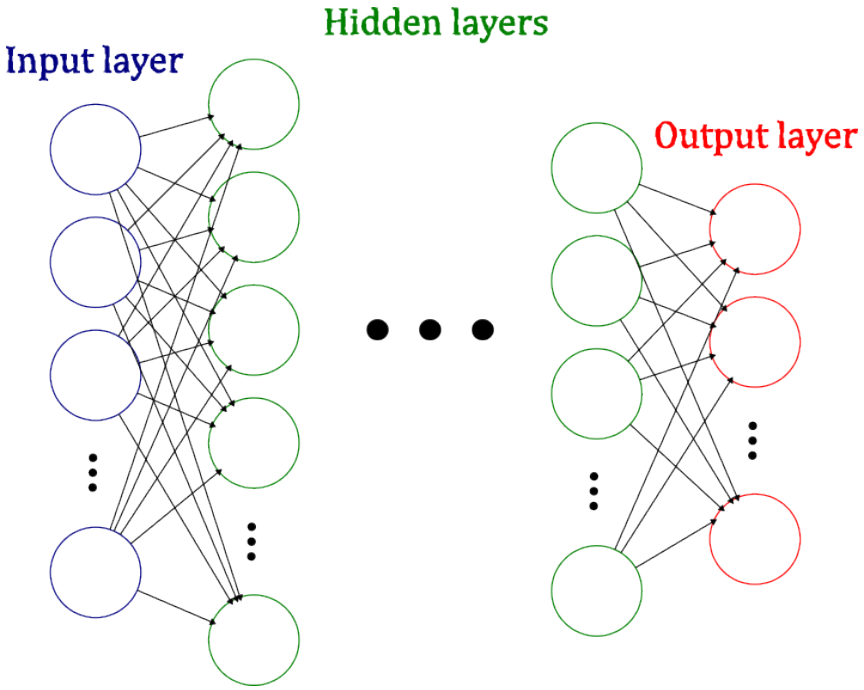


Figure 2.2: Depiction of the different layers in a neural network.

From figure 2.2 we clearly see the topology of a neural network, with the hidden layers being any layer that is not directly connected to the input nor the output.

The ANN is used to approximate an unknown function $\mathbf{F}(x)$. Say we have an ANN consisting of L layers, where from equation (2.4) we know that each neuron will act as a function f transforming the input they receive. Denoting f_i as all the neuron transformations present in a layer i. This means that, given an input $x^{(0)}$, the first layer will have the output $a^{(1)} = f_1(x^{(0)})$. The second layer will have the output $a^{(2)} = f_2(f_1(x^{(0)}))$. This continues recursively, so that the n'th layer will have the output $a^{(n)} = f_n(f_{n-1}(\dots(f_1(x^{(0)}))\dots))$. Formulating this for the ANN with L layers, we get

$$ANN(x^{(0)}) = f_{L+1} \circ f_L \circ \cdots \circ f_2 \circ f_1(x^{(0)}) = \mathbf{F}(x^{(0)}), \tag{2.5}$$

with each function f_i denoting the transformation at a respective layer i.

From section 2.1, we know that every neuron produces an output $a = \sigma(w^T x + b)$. This notation can be taken further to describe the output of every layer.

Denoting $x = [x_1, \dots, x_n]$ as an input vector consisting of all the nodal values in a layer k. Using the structure of figure 2.2, where every neuron in layer k is connected to every neuron in layer $k+1$.

Supposing there are n neurons in layer k, and m neurons in layer $k+1$. For every existing connection between any neuron in layer k and $k+1$, there must exist a weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$ that describe all the weight connections. The matrix product $x^T\mathbf{W}$ thus gives us a vector denoting the input of neurons with weights from layer k to layer $k+1$. At layer $k+1$ there must also be bias existing for every neuron, so that there will be a vector $\mathbf{b} = [b_1, \dots, b_n]$ consisting of the bias values. The matrix form for describing the input from layer k to layer $k+1$ will be

$$\sigma(x^T\mathbf{W} + \mathbf{b}) = \sigma(\mathbf{z}) = a^{(k)}, \tag{2.6}$$

where we use $\mathbf{z}$ as notation for the total linear output from a layer, and a^k as the activation from neuron k.

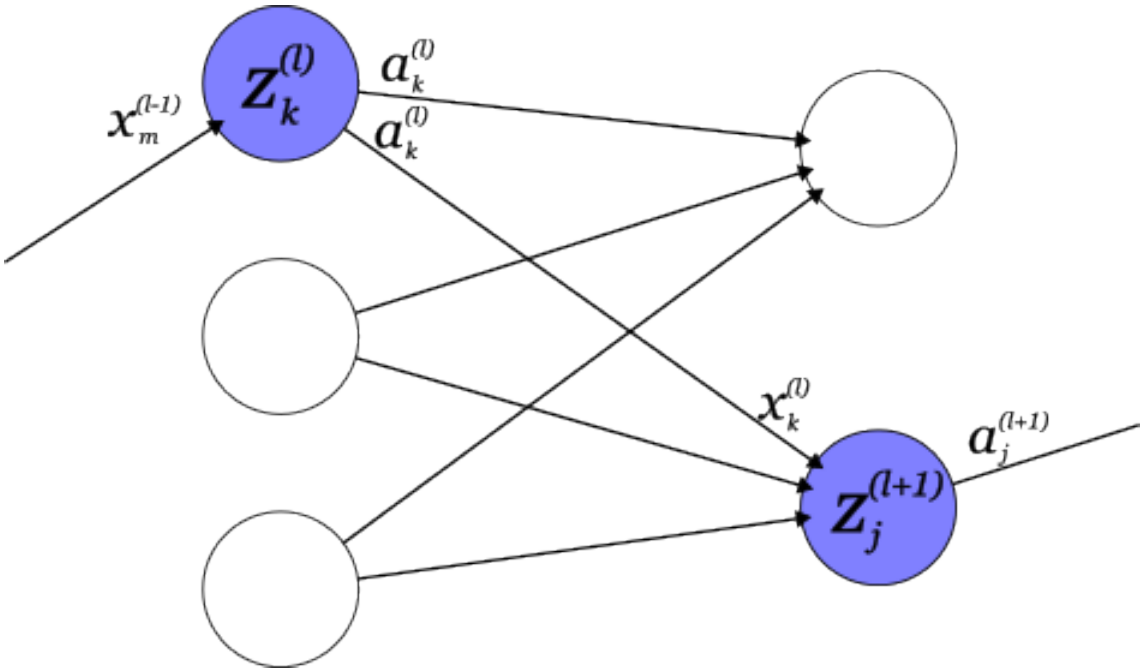


Figure 2.3: An illustration of the notation used for the different connections in a neural network.

Notationwise we will consistently write $w^{(l)}$ for the weight matrix at layer l, and $b^{(l)}$ for the bias at layer l. If the notations have subscripts, such as $w_j^{(l)}$ and $b_j^{(l)}$, this specifically concerns neuron j of the layer l. In those cases the weight is a vector and the bias is a scalar. This notation obviously holds for the input $x_j^{(l)}$, weighted input $z^{(l)}$, and output $x_j^{(l+1)}$ of a neuron.

2.3 Training and testing process

The datasets that are sent through neural networks are divided into two categories: a *training* and a *testing* set.

The training set is used for the initialization of the neural network. In order for the weights and biases to be set properly for the respective network, they need to "learn" from members of the same dataset. Therefore, the training set is passed through in a gradient-based process called *backward propagation*.

Accuracy is measured using *loss functions*. The loss function measures the difference between the actual labels and the output of the neural network. In order to enhance the performance, a gradient-based *optimizer* is used to minimize the loss.

The testing set is used to check accuracy after the training process. Distinguishing the testing set from the training set is important for measuring the neural network's performance using unbiased validation.

2.3.1 Loss functions

Loss functions are used to measure the accuracy, and are extremely important for optimization of weights and biases. The loss function takes the true labels y and the model output $\hat{y}$ as input values, and calculates the difference between the two. The loss function used is dependent on the problem that one aims to solve.

Introducing some loss functions that will be used for the purpose of this thesis.

Example 2.1 (Mean Squared Error)**.** *The Mean Squared Error is one of the more simple loss*

functions to understand.

Denoting N as the amount of samples in a dataset, we define the MSE-loss as

$$\mathcal{L}_{MSE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{2.7}$$

equivalent to the squared l_2-norm of the difference between y_i and $\hat{y}_i$. It is widely used in regression problems. A downside of this loss function is that it heavily penalizes outliers.

Example 2.2 (Negative Log-Likelihood Loss)**.** *Negative Log-Likelihood Loss is often used in classification problems, and is to be done after applying a probabilistic activation function.*

Denoting C as the amount of classes, i.e. the output of our respective model. The NLL-Loss for a single sample will be given as

$$\mathcal{L}_{NLL}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} t_{ji} \cdot \log \hat{y}_{ji}, \tag{2.8}$$

where t_{ji} is defined as a binary truth value that outputs 1 if y_j belongs to the i'th class, and 0 otherwise. As the output of the model has been sent through a probabilistic activation function, such as for example Softplus, the model output $\hat{y}_{ji}$ will be the probability for the variable being in the respective class.

The NLL-Loss thereby penalizes giving low probabilities to the correct classes.

Example 2.3 (Cross-Entropy Loss)**.** *The Cross-Entropy Loss, in this case for multiple classes, is a variant of the Negative Log-Likelihood Loss for the multi-class problem.*

Defined as

$$\mathcal{L}_{CE}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}\left[y_j \ln \hat{y}_j + (1 - y_j)\ln(1 - \hat{y}_j)\right], \tag{2.9}$$

we see that it, instead of discarding values through the binary truth value, it calculates the loss from all the probabilities given in the input. This means that the Cross-Entropy, in addition to penalizing low probabilities to the correct classes, also penalizes high probabilities to the wrong ones.

Example 2.4 (Kullback-Leibler Divergence Loss)**.** *Another name for the Kullback-Leibler Divergence Loss is relative entropy. Instead of looking directly at the input values of the model output y and the desired output, it computes a loss according to the difference between the respective probability distributions of the input values.*

Denoting $p(x)$ as the probability distribution for y, and $q(x)$ as the probability distribution for $\hat{y}$. The Kullback-Leibler Divergence is given as

$$D_{KL}(p|q) = \sum_{i=0}^{N} p(x_i)\log\left(\frac{p(x_i)}{q(x_i)}\right). \tag{2.10}$$

By looking at the probability distributions, we can calculate the amount of information lost when approximating one probability distribution with another.

Due to the loss functions measuring accuracy, one aims to keep it as low as possible. This means that updating the parameters W, b needs to be done whilst trying to minimize the loss. The optimization problem we are trying to solve during the training process is therefore given as

$$\min_{(W,b)}\left\{\frac{1}{n_{train}}\sum_{i=1}^{n_{train}}\mathcal{L}^i_{W,b}(y^i, \hat{y}^i)\right\}. \tag{2.11}$$

In order to perform gradient-wise operations on the loss functions, we depend on the following two assumptions:

1. The total loss will be the average of the loss functions of all inputs.
2. The loss function can be written as a function of the outputs.

Note that these assumptions hold for the loss functions already introduced in this thesis.

2.3.2 Backward propagation

Backward propagation refers to the scheme of updating the weights and biases according to the loss function. There are three key components to the backward propagation algorithm:

1. *Forward pass*: The input is propagated through the neural network. The loss function and the accuracy is calculated afterwards.
2. *Backward pass*: The loss is propagated backwards into the network, and the gradients of the loss with respect to the weights and biases are calculated.
3. *Weight and bias updates*: An optimization algorithm is used, using the gradients to update the weights and biases according to what will minimize the loss.

These steps repeat themselves for several iterations, or *epochs*. Note that it is common to send in the input in different *batches*, meaning that from n_s samples one may send in a set amount of $j < n_s$ samples at a time.

Forward pass We can use figure 2.3 to visualize the forward pass from a layer l to a layer $l+1$. This is the same process as explained in section 2.2, where the output of a layer k is sent to a layer $k+1$. The output from layer l is multplied with a matrix W featuring the weight components of all connections from layer l to layer $l+1$. All the nodal biases from layer $l+1$ are then added to the weighted output. The linear combination of the weights, inputs and biases is denoted as $z^{(l)}$. The full output at layer $l+1$ will be the activation of $z^{(l)}$, so that we use the notation $a^{(l+1)} = \sigma(z^{(l)})$ as the output.

The forward pass therefore consists of computing

$$x^{(l)} = \sigma(z^{(l)}) = \sigma(w^{(l)}a^{(l)} + b^{(l)}) = \sigma(w^{(l)}x^{(l-1)} + b^{(l)}) = a^{(l+1)} \tag{2.12}$$

for all layers $l = 1, \ldots, L$.

Backward pass: The whole aim of the backward pass step is to compute the gradient of the loss with regards to the weights and the biases. This is to see how changing the parameters of the neural network changes the accuracy, or the loss function.

Derivations of the backward pass formulas can be found in [43], [19] and [39]:

Recall the identities from section 2.2, where the weighted input to neuron j at the layer l is given as

$$z_j^l = \sum_{k=0}^{n} w_{jk}^l a_k^{l-1} + b_j^l. \tag{2.13}$$

and the activation of neuron j at layer l is given as

$$a_j^l = \sigma(z_j^l) \tag{2.14}$$

In order to derive the equations for $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$, we need to introduce the concept of an error δ_j^l in the j'th neuron of the l'th layer. The error is found by introducing a perturbation or change

in the input. The change in the input will lead to a change in the weighted input z_j^l, denoted as Δz_j^l. The shifted input will directly lead to a change in the activation, so that it becomes $a_j = l = \sigma(z_j^l + \Delta z_j^l)$. When this propagates through all the network layers, it will give a final change to the system of $\frac{\partial \mathcal{L}}{\partial z_j^l} \Delta z_j^l$.

This means that the error from the change in the system is given as

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l}. \tag{2.15}$$

We wish to find an expression for the error in the output layer of the entire system. Denoting the total amount of layers in the system as L. Using expression (2.15) with the chain rule applied, we get

$$\delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{2.16}$$

Recall from equation (2.14) that $a_j^L = \sigma(z_j^L)$. We can thus rewrite equation (2.16) so that

$$\frac{\partial \mathcal{L}}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L} \frac{\partial \sigma(z_j^L)}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L} \sigma'(z_j^L). \tag{2.17}$$

We thus have that the error of neuron j in the output layer L is given as

$$\delta_j^L = \frac{\delta \mathcal{L}}{\delta a_j^L} \sigma'(z_j^L). \tag{2.18}$$

Componentwise, $\frac{\delta \mathcal{L}}{\delta x_j^L}$ denotes the rate of change of the loss as a function of the j'th output neuron. Meanwhile, $\sigma'(z_j^L)$ denotes the rate of change of σ at z_j^L i.e. the weighted input of neuron j at the output (before going through the activation function).

The output error in layer L on matrix form is given as

$$\delta^L = \nabla_{a^L} \mathcal{L} \odot \sigma'(z^L), \tag{2.19}$$

where we have a vector of partial derivatives $\nabla_{a^L} \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial a_1^L}, \ldots, \frac{\partial \mathcal{L}}{\partial a_n^L} \right]$. Note that $\odot$ denotes the Hadamard product i.e. elementwise multiplication.

Another function we are interested in is the error in a layer l given in terms of the next layer, which would be layer $l+1$. Using the chain rule in vectorial form on equation (2.15) gives us

$$\delta_j^l = \frac{\delta \mathcal{L}}{\delta z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{2.20}$$

The term $\frac{\partial \mathcal{L}}{\partial z_k^{l+1}}$ is by definition equivalent to δ^{l+1}, so that we have

$$\delta_j^l = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta^{l+1}. \tag{2.21}$$

By the definition given in equation (2.13) we know that $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1}$. Furthermore, we know that $z_j^l = \sigma(z_j^l)$. The expression $\frac{\partial z_k^{l+1}}{\partial z_j^l}$ is thus reduced to

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l} \left(\sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} \right) = \frac{\partial}{\partial z_j^l} \left(\sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \right) = w_{kj}^{l+1} \sigma'(z_j^l), \tag{2.22}$$

where all the terms that had a $j \neq k$ and all bias terms were discarded under the derivation operation due to them being independent of the variable being differentiated.

Setting expression (2.22) into equation (2.21), we have that

$$\delta_j^l = \sum_k (w_{kj}^{l+1}\delta_k^{l+1})\sigma'(z_j^l). \tag{2.23}$$

This is given in vectorial form as

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l). \tag{2.24}$$

Taking the transpose of the weight w^{l+1} and multiplying it with δ^{l+1} is equivalent to moving the error backward through the network. Furthermore, taking the Hadamard product between this and $\sigma(z^l)$ is equivalent to moving the error back through the activation function in l.

The next equation we wish to derive is the rate of change of loss with respect to the bias. Constructing the expression for this and utilizing the chain rule for vectors gives us

$$\frac{\partial \mathcal{L}}{\partial b_j} = \sum_k \left(\frac{\partial \mathcal{L}}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_j} \right). \tag{2.25}$$

Note that $\frac{\partial z_k^l}{\partial b_j^l} = 0$ when $j \neq k$ due to independence. This gives us

$$\frac{\partial \mathcal{L}}{\partial b_j} = \frac{\partial \mathcal{L}}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j} = \delta_j^l \frac{\partial z_j^l}{\partial b_j}, \tag{2.26}$$

where the definition of δ_j^l has been used to simplify the expression.

Using the fact that $z_j^l = (\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$, we see the only term in $\frac{\partial z_j^l}{\partial b_j^l}$ that will not equate zero when differentiated with respect to the bias is in fact the bias. Furthermore, all the biases are independent of each other, meaning we have

$$\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial}{\partial b_j^l}(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) = \frac{\partial b_j^l}{\partial b_j^l} = 1. \tag{2.27}$$

The equation for the rate of change of the loss function with respect to the bias is therefore given as

$$\frac{\partial \mathcal{L}}{\partial b_j} = \delta_j^l. \tag{2.28}$$

Physically this makes sense as the bias is not related to the different inputs, but the neuron in itself. In matrix form this can be written as

$$\frac{\partial \mathcal{L}}{\partial b} = \delta. \tag{2.29}$$

The last equation we want to derive is the rate of change of the loss with respect to the weights. Constructing this expression and using the chain rule, we have

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \sum_m \left(\frac{\partial \mathcal{L}}{\partial z_m^l} \frac{\partial z_m^l}{\partial w_{jk}^l} \right). \tag{2.30}$$

Again, terms of the sum disappear when $m \neq j$ due to independence between neurons, leaving us with

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l}. \tag{2.31}$$

Using the expression $z_j^l = \sum_k (w_{jk}^l a_k^{l-1} + b_j^l)$ to find $\frac{\partial z_j^l}{\partial w_{jk}^l}$, we get

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l} \sum_m (w_{jm}^l a_m^{l-1} + b_j^l) = \frac{\partial w_{jk}^l}{\partial w_{jk}^l} a_j^{l-1} = a_j^{l-1}, \tag{2.32}$$

where again terms disappear due to independence when $k \neq m$.

The equation for the rate of change of the loss with respect to the weight is therefore given as

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{2.33}$$

The expression looks complicated due to the weight matrix being multi-indexed. What the expression is saying is that the derivative of any connection between the neuron k in layer $l-1$ and neuron j in layer l will be the product of the input in the connection (i.e. neuron k in layer $l-1$) multiplied with the error of the output (i.e. the error of neuron j in layer l).

We have thus derived the fundamental equations of backward propagation:

$$\begin{aligned} \delta^{(L)} &= \nabla_a \mathcal{L} \odot \sigma'(z^{(L)}), \\ \delta^l &= ((w^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)}), \\ \frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} &= a_k^{(l-1)} \delta_j^{(l)}, \\ \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} &= \delta_j^{(l)}. \end{aligned} \tag{2.34}$$

From equations 2.34 we have expressions that can be used to optimize the weights and biases of the network. The relevance of them can be seen more clearly in the algorithm for backward propagation:

Algorithm 1 Backward propagation

Pass $x^{(0)}$ to the first layer of the network. ▷ Input
Compute $a^{(1)} = \sigma(w^{(1)} x^{(0)} + b^{(1)})$
for each layer $l = 2, \ldots, L$: **do** ▷ Forward pass
    $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$
    $a^{(l)} = \sigma(z^{(l)})$
end for
Compute $\delta^{(L)} = \nabla_{a^L} \mathcal{L} \odot \sigma'(z^{(L)})$ ▷ Output error
for each layer $l = L-1, L-2, \ldots, 2$: **do** ▷ Backward pass
    $\delta^l = ((w^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)})$
end for
Compute $\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}$ ▷ Computing the output
Compute $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}$
return $\frac{\partial \mathcal{L}}{\partial w^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(l)}}$ ▷ To be used in an optimzation algorithm

2.3.3 Optimization algorithms

Taking the derivatives calculated as inputs, the optimization schemes are crucial for finding the best possible combination of bias and weights. There is an abundance of schemes to choose between, as well as parameters that can be tweaked.

As stated in [26], there are some hyper parameters that will are relevant in all optimization algorithms:

- *Learning rate*: The learning rate, denoted α for the sake of this thesis, is defined as the step size for the parameter update. It has a big influence on the convergence of the optimization

algorithm: if the learning rate is too small there will be very slow convergence, and the scheme may even get stuck in a bad local minima. However, if the learning rate is too large the optimization scheme may diverge. Some schemes implement *adaptive learning rates*, where the learning rate may change throughout the iterations of the algorithm.

- *Batch size*: The batch size is defined as the number of inputs sent through the model during an iteration, and so in an optimization scheme will also affect the amount of gradients calculated at any given time. Some optimization schemes use the entire training set, however this requires a large amount of memory and may not be feasible for many computers. Other schemes use *mini-batches*, where the samples are sent in one-at-a-time. The batch size also determines how often the model parameters are updated, where a small batch size will have fewer computations and therefore more frequent updates.
- *Convergence criteria*: The convergence criteria is used for deciding when to stop the training process. In many cases, a set amount of iterations, or *epochs*, is chosen beforehand. It is also possible to have a loss value to stop at, in order to prevent over-fitting.

With this in mind, we can introduce some of the optimization schemes that will be used in the scope of this thesis. For notations' sake, the parameters that will be updated (i.e. $W^{(l)}$ and $b^{(l)}$) will be denoted θ. This is done with the aim of generalization. Furthermore, when choosing one scheme both variables are updated in the same manner, meaning there is no need for double notation. Additionally, it is already implied that the parameter within every layer i.e. weights and biases need to be updated, meaning there is no need for layer specifications.

Note that all the schemes are implemented within iterations ranging from $t = 0, 1, \ldots, n_{epochs}$. We therefore have that $\theta_t = \theta(t) = (W(t), b(t))$, where t denotes the iteration.

Example 2.5 (Gradient Descent (GD))**.** *Gradient descent is the most fundamental optimization scheme. The scheme is given as*

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t). \tag{2.35}$$

As one can see in the equation, the parameters θ are adjusted at every iteration according to the opposite direction of the gradient of the loss. The role of the learning rate α is pretty clear here, giving weight to how big of an influence the gradient of the loss will have.

Note that Gradient Descent bases itself on the whole dataset being sent in as a batch.

Example 2.6 (Stochastic Gradient Descent (SGD))**.** *Stochastic Gradient Descent is adaption of regular Gradient Descent, but with the usage of mini-batches. Therefore, the scheme is instead given as*

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t; x^{(i)}; x^{(i)}). \tag{2.36}$$

Due to the usage of mini-batches, the updates are more frequent within SGD - actually there is an update for every sample in the training set. The convergence is usually quicker as a result, as well as less memory being used per iteration.

Example 2.7 (Stochastic Gradient Descent with Momentum)**.** *Momentum is a technique that can be used within optimization schemes. Instead of only basing itself on the gradient in the current step, momentum accumulates gradients of past steps in order to determine the direction to go [27].*

One chooses a momentum coefficient η in the range $[0, 1]$ that determines the importance of past gradient directions.

The scheme is on the form

$$\begin{aligned} v_{t+1} &= \eta v_t - \alpha \nabla_\theta \mathcal{L}(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned} \tag{2.37}$$

This means that the retained gradient multiplied with the momentum coefficient is used for electing parameters.

Adding momentum to the scheme can both help overcoming local minima, as well as accelerating convergence. However, it does mean that there is yet another hyper parameter to tune.

Example 2.8 (Adaptive Moment Estimation (Adam))**.** *The Adam scheme combines an adaptive learning rate with the advantages of using momentum within a scheme.*

The scheme is given as[24]:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t), \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2, \\ \theta_t &= \theta_{t-1} - \alpha \frac{m_t}{(1-\beta_1^t)(\sqrt{v_t(a-\beta_2^t)^{-1}} + \epsilon)}. \end{aligned} \tag{2.38}$$

The hyper parameters β_1 and β_2 are exponential decay rates and so are in the domain $[0, 1]$. The parameter ϵ is a small number added to avoid division by zero.

The Adam optimization scheme has generally good convergence properties, however it does use a lot of storage as well and therefore requires a lot of memory.

2.4 Relevant layer types

The layers of an artificial neural network perform transformations to the input, reducing or increasing the dimensions of the data. The different types of layers will perform different types of transformations.

Additionally, there are also some enhancements that determine the dimensional output of the layer when performing a transformation.

Note that the terms *filter* and *kernel* are frequently used and have the same definition. These are small matrices used on input data to commit transformations or feature extractions.

2.4.1 Enhancements for ANN layers

The *stride* is defined as the amount of cells that the filter moves by per operation. The difference in output of a filter is visualized in figure 2.4.

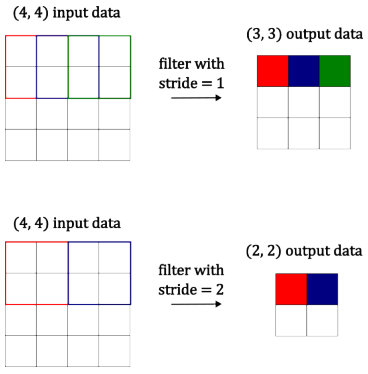


Figure 2.4: The difference in output of a filter, dependent on the stride.

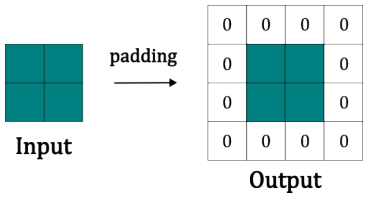


Figure 2.5: Padding demonstrated on an input of size (2,2). Note that this is a padding of size 1.

Padding is a technique used in order to preserve spatial dimensions and prevent information loss. When using padding, we are essentially adding zeros to the edges of the input before performing feature extraction. The usage of padding thus gives a larger dimensional output, as it gives us prominent features from edges and corners. This is demonstrated in figure 2.5, where a padding of size 1 is applied to an input matrix of size (2,2). Note that it is possible to increase the padding. The padding size denotes the amount of rows and columns of zeros to add to the image.

Weight decay, otherwise known as regularization, is the action of adding a penalty term λ to the loss function. Given a loss function $\mathcal{L}(\theta)$, we thereby denote the new loss function with weight

decay as

$$\mathcal{L}_{wd}(\theta) = \mathcal{L}(\theta) + \lambda||w||, \tag{2.39}$$

where the norm can be chosen from preferences, but is commonly the L_2 or L_1 norm. The addition of the norm of the weights multiplied with the decay encourages the model to minimize their weights. Ultimately the goal is to prevent over-fitting, as it is penalized when a model is overly complex or dependent on a dominating set of weights.

Dropout is an action that is implemented in the beginning of a layer. It sets a random fraction of the input to zero during each training iteration, which should improve generalization and prevent over-fitting. The dropout fraction can be chosen according to preferences.

When we implement *batch normalization*, it is usually done for the input of every single layer. Batch normalization is normalizing the inputs of each layer so that they have mean $\mu = 0$ and standard deviation $\sigma = 1$. This implies that every input follows the normal distribution. Batch normalization reduces the importance of having fitting initial values (as they all will be scaled equally regardlessly), and encourages the model to focus on trends rather than memorizing training samples.

Data augmentation is a technique for making the model more robust by preventing over-fitting and improving the generalization[50]. The aim is to augment parts of the training data whilst preserving the respective labels. Some examples of augmentation could be horizontal flipping, random cropping, color transformations, and so on. This allows the model to train on a wider variation of data, and thereby recognize more features or patterns.

2.4.2 Fully connected layers

Fully connected layers are layers where every neuron in one layer is connected to every neuron in the subsequent layer. This is depicted in figure 2.2.

Using the expression of the forward pass from equation 2.12, the output from a fully connected layer l would be on the form

$$x^{(l)} = \sigma(w^{(l)}x^{(l-1)} + b^{(l)}). \tag{2.40}$$

Fully connected layers typically expect one-dimensional arrays as inputs. This means that if fully connected layers are connected with other layers that deal with multi-dimensional data, the input to the fully connected layer would need to be flattened i.e. mapped to a one-dimensional array.

2.4.3 Pooling layers

The pooling layer performs a dimensionality reduction and feature extraction. The aim is to downsample the spatial dimensions while still retaining the most important information. Another purpose is to introduce translational invariance, which is when the system produces the same output regardless of shifts in the input.

Note that pooling operates independently on the different channels in the layers. This means that if one were to send in a colored picture, the pooling operations would happen on the Red, Green and Blue channels independently. Thus, the amount of channels stays the same despite the downsampling.

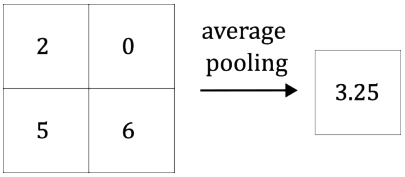


Figure 2.6: Demonstration of average pooling.

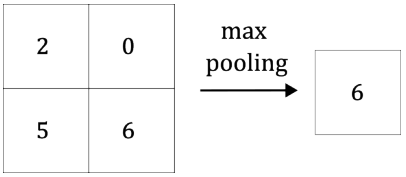


Figure 2.7: Demonstration of max pooling.

In figure 2.6, average pooling is shown visually. Here we see how, given a 2x2 input matrix, the input is downsampled to a single value that describes the features of the input. On the other hand, in figure 2.7 the same input is downsampled using max pooling. Although both methods are efficiently downsampling, they may lead to very different results dependent on the difference between the input values.

Max-pooling captures more dominant features, and is therefore useful for cases where it is important to capture the most prominant features in a specific region. This is therefore widely used in object recognition or pattern detection. Average pooling, on the other hand, summarizes the local spatial information whilst giving a smoother representation. It is often used in the final stages of networks to gain a global representation of the features extracted from other layers.

2.4.4 Convolutional filters

The aim of the convolutional layer is to extract features using convolution. Essentially the idea is to apply a transformation to the input. This is done through applying a *kernel*, i.e. a small matrix consisting of weights, to the input.

Mathematically, a convolution is defined as a mathematical operation between two functions that produce a third one. As the convolution operation is shifting the one function, g, over to the other, f, the third integral produced (i.e. the convolution) will reflect how much they overlap[61]. The convolution between f and g is given as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau. \tag{2.41}$$

The inputs in neural networks are generally multi-dimensional arrays, and so we are more interested in the equation for convolution in its discrete form:

$$(f * g)(x) = \sum_{-\infty}^{\infty} f(\tau)g(t-\tau). \tag{2.42}$$

The two tensors that are used in a convolution process are, as stated, the input tensor I and a kernel K. The input tensor is typically multi-dimensional, meaning one needs to apply convolution to several axis at the same time. We thereby have

$$(I * K)(i,j) = \sum_{\alpha}\sum_{\beta} I(\alpha,\beta)K(i-\alpha, j-\beta) \tag{2.43}$$

as the equation for the convolution between I and K. This can be rewritten as

$$(I * K)(i,j) = \sum_{\alpha}\sum_{\beta} I(i-\alpha, i-\beta)K(\alpha,\beta), \tag{2.44}$$

due to convolution being commutative.

Changing the subtraction symbols to addition, we have the cross-correlation function

$$(I * K)(i,j) = \sum_{\alpha}\sum_{\beta} I(i+\alpha, i+\beta)K(\alpha,\beta). \tag{2.45}$$

Equation 2.45 is from where the convolutional filters in neural networks are actually based[19]. The shift in the sign means that the focus is on the similarities between I and K when the input I is shifted relative to the kernel K.

In figure 2.8 we visualize the effects of a convolutional kernel being applied on some input. By applying the kernel, we can decrease the dimensions of an image and bring all relevant information into a single feature.

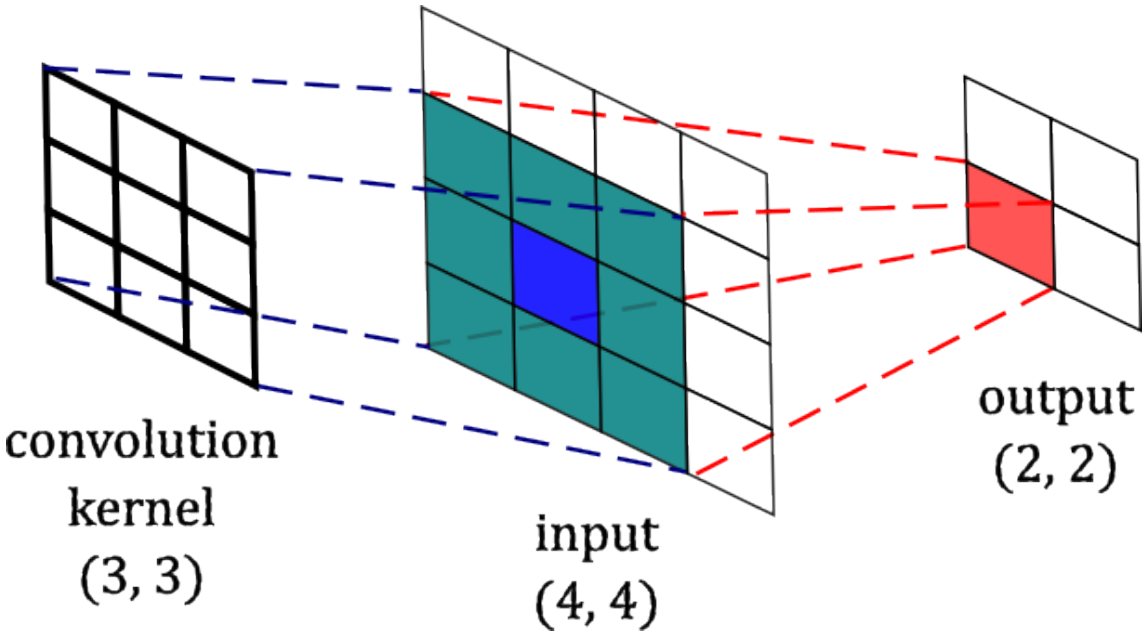


Figure 2.8: Visualization of a discrete 2d conovolutional layer, which is a more common one for image recognition. The convolution kernel compresses the values of the input-data into a smaller output, extracting features from neighboring nodes.

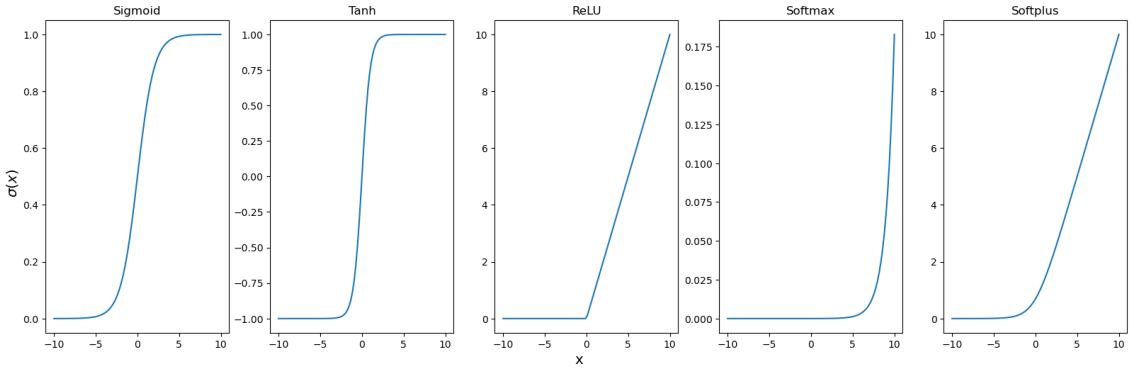


Figure 2.9: Various famous activation functions in the domain [-10,10].

2.4.5 Activation functions

Activation functions[52], as introduced in section 2.1, introduce non-linearity to the layers of the neural network. They can be incorporated into convolutional layers, or be used for classification in the final layers of a network.

There are many different kinds of activation functions, and the type of activation function to be used depends on the given task.

Example 2.9 (Sigmoid function)**.** *The Sigmoid function is otherwise known as the logistic function. It is a widely employed activation function.*

The Sigmoid function is given as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.46}$$

Inputs to the activation function that are much smaller than -1 will be mapped to 0, whilst inputs much larger than 1 will be mapped to 1. Inputs that are in between this domain will be mapped somewhere in between 0 and 1.

Example 2.10 (Tanh function)**.** *The Tanh activation function is given as*

$$\sigma(x) = \tanh(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.47}$$

The Tanh activation function works in a similar manner to the Sigmoid function, with an "S" shape in the primary domain that it's based.

Example 2.11 (ReLU function)**.** *The Rectified Linear Unit function (ReLU) is a piecewise linear activation function.*

Given as

$$\sigma(x) = \max(0, x), \tag{2.48}$$

ReLU directly provides x provided x is positive. Otherwise it returns 0.

Example 2.12 (Softmax function)**.** *Softmax is a probability-based activation function. It provides a value based on the entire input, meaning an arbitrary value a gets a different output dependent on the other input values. It is given as*

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}. \tag{2.49}$$

The $\sigma(x)$ values for the softmax function in figure 2.9 are much smaller than for the other activation functions. The range for this activation function is indeed [0,1], however it works as a probability distribution. This means that, for a given input $x = [x_1, \ldots, x_n]$ we will have $\sum_{i=1}^{n} \sigma(x_i) = 1$.

Example 2.13 (Softplus function)**.** *The Softplus activation function is a smooth continuous version of ReLU, and is given as*

$$\sigma(x) = \log(1 + e^x). \tag{2.50}$$

Note that the function has similarities to Sigmoid. As $x \to -\infty$, they are in fact the same function.

2.5 Feed-Forward Neural Networks

Feed-forward Neural Networks are the most simple kind of neural networks, and serves as a building block for many other neural network architectures.

Their most important feature is that their layers are only connected in one direction, from input to output. Each layer provides as output a transformation to the input, as explained in section 2.2. Each parameter i.e. weight or bias is used exactly once in the network, as there are no recurring layers. Every neuron is variable i.e. the amount of hidden layers and neurons per layer is flexible.

Defining an FNN consisting of L hidden layers, with an output from the network given as $\hat{y}$. From the recursiveness of the functions of the various layers, we have that

$$\begin{aligned}\hat{y_j} = \sigma(z^{(L)}) = \sigma\left(\sum_{i=0}^{n_L} w_{ij}^{(L)} x_i^{(L-1)} + b_j^{(L)}\right) = \sigma\left(\sum_{i=0}^{n_L} w_{ji}^{(L)} \sigma\left(\sum_{q=0}^{n_{L-1}} w_{iq}^{(L-1)} x_q^{(L-2)} + b_i^{(L-1)}\right) + b_j^{(L)}\right) \\ = \cdots = \sigma\left(\sum_{i=0}^{n_L} w_{ij}^{(L)} \sigma\left(\sum_{q=0}^{n_{L-1}} w_{jq}^{(L-1)} \sigma\left(\ldots\left(\sigma\left(\sum_{p=0}^{n_1} w_{sp}^{(1)} x_p^{(0)} + b_s^{(1)}\right) + b_r^{(2)}\right)\ldots\right) + b_j^{(L-1)}\right) + b_i^{(L)}\right),\end{aligned} \tag{2.51}$$

where $\hat{y}_j$ denotes the output from the neuron j at the output layer.

Due to all the neurons being interconnected and independent, all the parameters are independent and none are shared. This means that deep Feedforward Neural Networks have a lot of parameters, which may be computationally expensive during the training process as well as susceptible to overfitting.

FNNs are widely used for classification problems, both on their own and in combination with other layers. It is viable, in classification tasks, to for example plant them at the end of the network to do the sorting and classification.

2.6 Convolutional Neural Networks

Convolutional Neural Networks are a subgroup of FNNs. This means they have the same structure, following one direction from input to output. CNN's are specifically designed to receive grid-like data, making them suitable for tasks such as image processing.

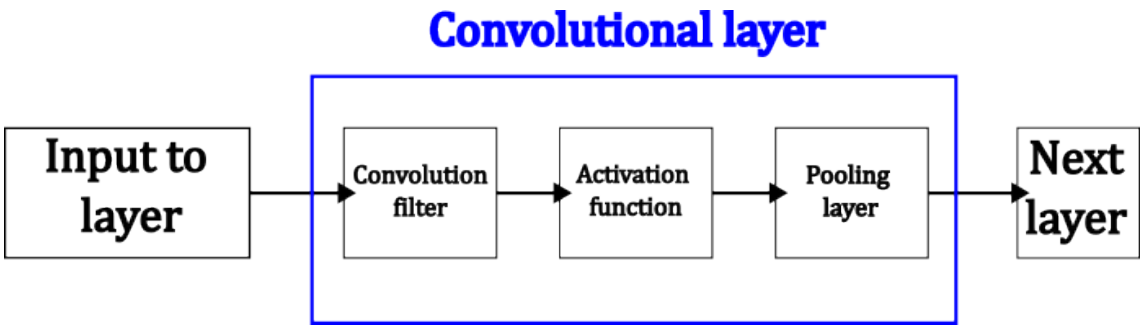


Figure 2.10: The architecture of a convolutional layer.

A CNN is defined as a Neural Network that consists of at least *one* convolutional layer. As shown in figure 2.10, the convolutional layer consists in fact of three layers: a convolution filter, an activation function, and a pooling layer. This architecture is so common that one groups the combination of the three together under this term.

The convolutional layer has three advantages over other layer types[19, 14]:

- *Sparse connectivity*: The usage of convolutional kernels that are smaller than the input means that the output of the convolutional layer will only be connected to a small, local field of the input. The fact that the output is only dependent on a small amount of inputs is called sparse connectivity. This leads to fewer operations for getting the output and more storage space.
- *Parameter sharing*: The kernels in convolutional layers are generally applied multiple times to all the different sections of the input data. The essence is that the feature detector useful in one part of the input is useful in other parts of the input. The reusage of the kernel over the entire input significantly reduces the amount of parameters needed, and is what we call parameter sharing.
- *Translation invariance*: Translation invariance signifies that a shift in the input doesn't affect the output. The aim here is that, in spite of the shift, similar features will still be detected. The convolutional layer has this advantage due to the same filter and thus the same specific feature detector being applied everywhere. This makes the convolutional layer more robust.

Due to the CNNs having at least one convolutional layer, they all inhibit the same advantages.

The general architecture of CNNs is portrayed in figure 2.11. Here we see how the input is generally sent through convolutional layers and pooling layers for feature extraction. After sufficient feature extraction has taken place, dependent on the network architecture, the data is flattened before being sent through a FNN. The FNN ultimately gives its predictions as its output.

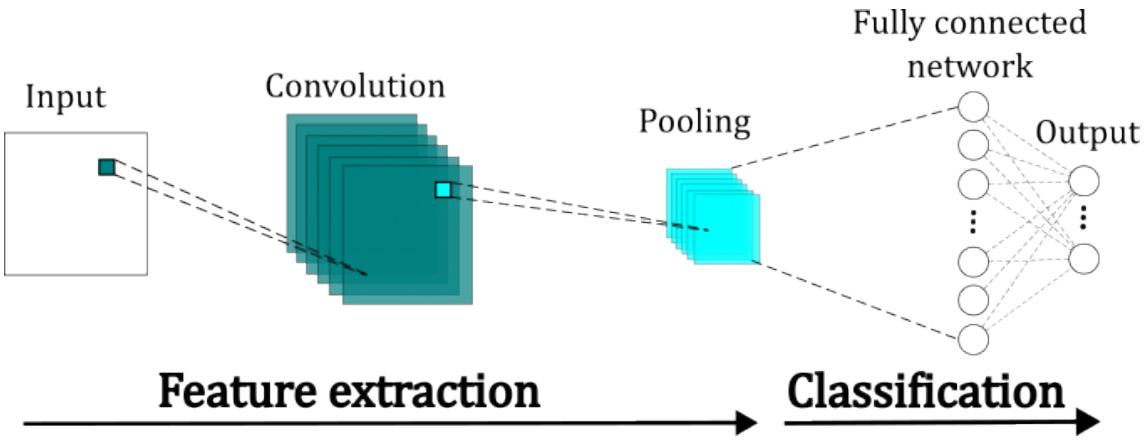


Figure 2.11: The architecture of a Convolutional Neural Network.

3 Mathematical preliminaries

The concepts presented here build an important foundation regarding spectral decomposition and solutions to optimization problems. They are not directly related to model order reduction methods. However, they play a big role in the reduction methods presented for the scope of this thesis, making them necessary preliminaries.

3.1 Least Squares Method (LSM)

Given some independent variables $x = [x_1, x_2, \ldots, x_n]$ and corresponding observations $y = [y_1, y_2, \ldots, y_n]$, a common problem posed is how to best fit a model to the data. The method of least squares provides a solution to this issue that can be applied to both linear and non-linear data [5].

Introducing the best fitted model as

$$\eta = f(x, \beta), \tag{3.1}$$

where β denotes the parameters that are to be estimated in order to give the best possible fit.

In experiments, it is assumed that observations can be affected by random errors. This means that the observations can be expressed as

$$y_i = \eta_i + e_i, \tag{3.2}$$

where e_i denotes the residual error. Essentially, equation (3.2) tells us that the observations will be a sum of the residual error and the underlying value of the response η_i.

In order to have an output that is as accurate as possible, one aims to minimize the residual error. The least squares method bases itself on the best estimates being those that minimize the sum of the squared residuals, solving

$$\min_{\beta} \sum_{i} (e_i)^2 = \min_{\beta} \sum_{i} (y_i - \eta_i)^2. \tag{3.3}$$

This can also be seen as minimizing the sum of the squared differences between the true and the calculated output.

Given a linear system of equations $Ax = b$, where we have $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^m$, and $n > m$. Let us say A is singular, meaning there exists no inverse for A. The solution of the system of equations can be, in this case, found through least squares method.

The residual of the system is given as $r = Ax - b$. The least squares method is based on minimizing the residual, i.e. it solves the optimization problem

$$\min_{x \in \mathbb{R}^{>}} ||Ax - b||_2, \tag{3.4}$$

and is the optimal solution to the minimization problem.

Proof. The length or inner product of the residual is denoted as

$$r^T r = (b - Ax)^T (b - Ax) = (b^T - x^T A^T)(b - Ax),$$

and the aim is to minimize the length of the residual with x as subject to change.

By differentiating the inner product of the residual with respect to x, we get

$$\frac{\partial (r^T r)}{\partial x} = 2A^T(b - Ax) = 2A^T b - 2A^T Ax. \tag{3.5}$$

Setting this to zero implies that we have a local optimum of the error x^*, and gives us the relationship

$$A^T b = A^T A x^*. \tag{3.6}$$

Assuming that $A^T A$ has an inverse, we left multiply this on both sides of the equation in order to gain the optimum

$$x^* = (A^T A)^{-1} A^T b, \tag{3.7}$$

which is otherwise known as the method of least squares. □

Note that, in the cases where $A^T A$ does in fact not have an inverse, it is still possible that x^* has an optimal parameter that minimizes the residual, but this needs to be found in that case through methods such as singular value decomposition or finding the pseudoinverse.

3.2 Singular Value Decomposition (SVD)

The following section is taken directly from [4].

Given a matrix $\mathbf{Y} \in \mathbb{R}^{m \times n}$, the singular value decomposition[2] allows us to factor the matrix $\mathbf{Y}$ into a product of three matrices $\mathbf{U}\Sigma\mathbf{V}^*$ instead, where $\mathbf{U} \in \mathbb{R}^{m \times m}$, $\Sigma \in \mathbb{R}^{m \times n}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$.

The columns of $\mathbf{U}$ and $\mathbf{V}$ are in fact the left and right eigenvectors of $\mathbf{Y}$ respectively. Additionally, these are unitary orthogonal matrices. This means that they have the property

$$UU^* = U^*U = I, \quad VV^* = V^*V = I.$$

Our matrix Σ consists of the singular values of $\mathbf{Y}$ on the diagonal, and 0 elsewhere. Mathematically this is expressed as

$$\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_p), \quad p = \min\{m, n\}.$$

The singular values will be arranged in descending order, meaning $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p$.

Matrices describe linear transformations, and what the SVD actually does is to break the action of matrix Y into three parts. The matrix Σ denotes the stretch or compression, whilst our matrices $\mathbf{U}$ and $\mathbf{V}$ denote rotations. Therefore, the singular values in Σ actually give us information regarding the significance of the various rotations within the linear system.

From elementary matrix operations, we know that

$$\mathbb{R}^{m\times r} \times \mathbb{R}^{r\times r} \times \mathbb{R}^{r\times n} \to \mathbb{R}^{m\times n}.$$

This provides an important idea for matrix operations: we can truncate our matrices gained from singular value decomposition into $\tilde{\mathbf{U}} \in \mathbb{R}^{m\times r}$, $\tilde{\Sigma} \in \mathbb{R}^{r\times r}$, and $\tilde{\mathbf{V}} \in \mathbb{R}^{r\times n}$, and their product will still be the matrix $\mathbf{Y} \in \mathbb{R}^{m\times n}$.

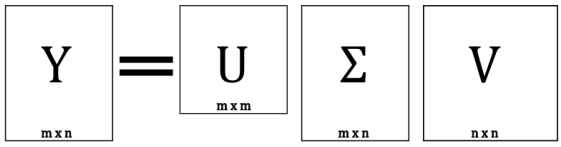


Figure 3.1: Illustration of singular value composition for an arbitrary matrix Y.

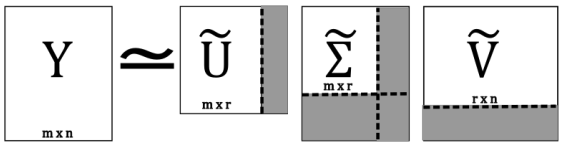


Figure 3.2: Illustration of singular value composition for an arbitrary matrix Y, with reduction. The grey area marks the parts of the matrices that are cut off.

From the singular value decomposition we can thereby approximate matrices using a lower rank.

Proposition 1. *Given a matrix $A \in \mathbb{R}^{m\times n}$, the lower-rank matrix $A_k = U_k\Sigma_k V_k^T$ found from the singular value decomposition is optimal.*

Proof. The idea behind this proof is found in [3] and [60]. Mathematically, we want to show that

$$||A - A_k||_F^2 = \min_{B\in\mathbb{R}^{d\times k}, C\in\mathbb{R}^{k\times n}} ||A - BC||_F^2. \tag{3.8}$$

We assume that B is fixed as the optimal orthonormal matrix. Given this assumption, what would the optimal C be?

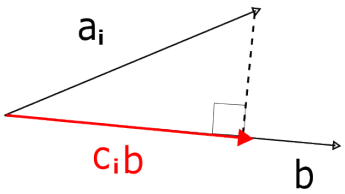


Figure 3.3: Visualization of the projection of an element a_i from A onto a column of B, in the 1D case. c_i needs to be chosen so that $c_i b$ will be the respective projection.

Geometrically, the optimal C would be the projection of A onto b. This is visualized for the 1-dimensional problem in 3.3, where one can clearly see that the minimum distance between any a_i and $c_i b$ would be the fine line denoting the projection $c_i = \langle a_i, b\rangle$.

We therefore have that $C = B^T A$. Our problem can thus be rewritten as

$$\min_{B\in\mathbb{R}^{d\times k}, BB^T=I} ||A - BB^T A||_F^2. \tag{3.9}$$

We know that $||(A - BB^T A) + BB^T A||_F^2 = ||A||_F^2$.

The matrix Pythagorean theorem states that if we have two mutually orthogonal matrices M and N so that $M^T N = 0$, then $||M + N||^2 = ||M||^2 + ||N||^2$ holds.

Using this quality, we have that

$$||A - BB^TA + BB^TA||_F^2 = ||A - BB^TA||_F^2 + ||BB^TA||_F^2.$$
$$\Rightarrow ||A - BB^TA||_F^2 = ||A||_F^2 - ||BB^TA||_F^2,$$

which means that our optimalization problem can be rewritten as

$$\min_{B\in\mathbb{R}^{d\times k}, BB^T=I} ||A||_F^2 - ||BB^TA||_F^2. \tag{3.10}$$

Obviously, with a set A, this cost function would be at its lowest when $||BB^TA||_F^2$ is largest possible. Therefore, this is equivalent to

$$\max_{B\in\mathbb{R}^{d\times k}, BB^T=I} ||BB^TA||_F^2. \tag{3.11}$$

We rewrite A to its singular value decomposition, so that our problem is restated as

$$\max_{B\in\mathbb{R}^{d\times k}, BB^T=I} ||BB^TU\Sigma V^T||_F^2. \tag{3.12}$$

As B and V both are orthonormal by definition, multiplying on the left hand side of the norm with B and the right hand side with V^T will just change the basis in a norm-preserving manner. This means that the left hand side multiplication of B and right hand side multiplication of V^T can be omitted from the optimization problem, so that we restate the problem as

$$\max_{B\in\mathbb{R}^{d\times k}, BB^T=I} ||B^TU\Sigma||_F^2. \tag{3.13}$$

B^T and U are both orthonormal, meaning $Q = B^TU$ will have orthonormal columns, and so the norm of these cannot be greater than 1. This means that the sum of all the norms of Q's columns will be at most k.

We therefore know that

$$||Q\Sigma||_F^2 = \sum_i ||q||_i^2\sigma_i^2 \leq \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_k^2.$$

Maximizing $||B^TU\Sigma||_F^2$ would therefore be at most the sum of all the singular values squared. Remember that U is orthonormal; when $B = U_k$, we obtain the maximum as it would give us the sum of all singular values squared up to the point of truncation.

□

So, it is thus proven that the optimal lower-rank matrix for a matrix A would be the truncated SVD, showing the importance of utilizing exactly this approximation in numerical methods.

In order to show how the singular values can be of different importance, we have made a visualization of the singular values of a dynamical system, with the amount of timesteps it has been integrated varying.

Figure 3.4 shows how the singular values of a system vary, on a logarithmic scale, from the first singular value to the last. In order to make this consistent, the same inital values y are used, however the amount of timesteps used to integrate it varies, thus giving a variation in the column-amount. The opacity of the colors on the graph denote the difference in total amount of columns.

In this case, our chosen system consists of 100 particles. We clearly see that the worth of the singular values will vary - especially when there are few columns taken into consideration. This makes sense from a physical point of view, as less columns gives us less information about the

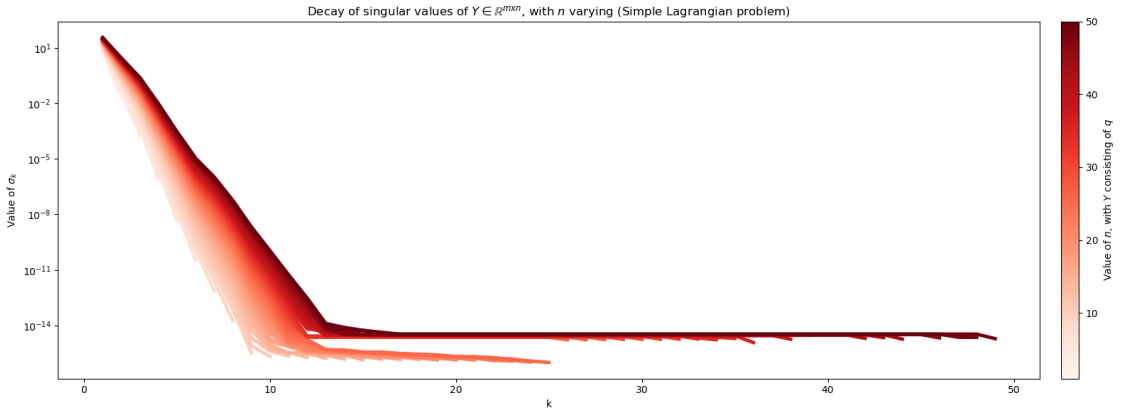


Figure 3.4: Singular values of varying amount of columns, using Y from example 1.

system. The lines of greater opacity, on the other hand, are closer to a set value. This signifies that as long as the amount of columns we have is sufficient, the singular values will converge to some set values. Furthermore, we see that for all cases there is a quick decay in the values. Even in the case of 50 columns, there are only ~ 13 singular values of a size larger than 10^{-14}, showing that an increased amount of time steps will not lead to more singular values of "importance".

So to sum up having a *sufficient* amount of columns is the most important when finding the singular values of a system such as the one in problem 1. As long as this is done, if we are truncating U based on the value of σ_k the amount of singular values that we "keep" will be around the same regardlessly.

3.3 Frequent Directions Method

Frequent Directions method (FD) was first proposed by [30] in 2013 as a matrix sketching procedure, and later revised in [18]. A matrix sketching procedure is a scheme where a matrix $A \in \mathbb{R}^{n \times m}$ of large dimensions is approximated using a smaller matrix $B \in \mathbb{R}^{l \times m}$, where $l << m << n$. The sketching in FD is done using a *streaming process*, where we iterate through the rows of A one at a time.

Frequent Directions aims to maintain the best rank-l approximation of A[41]. Recall from section 3.2 that the best rank-l approximation is the truncation of the SVD of matrix A. Generating the SVD can be extremely computationally expensive when handling matrices of large dimensions. FD therefore generates an approximation of the truncated SVD, and thereby finds the best lower-rank approximation to A.

Denoting the rows of $A = [a_1, a_2, \ldots, a_n]^T$, and the rows of $B = [b_1, b_2, \ldots, b_n]^T$, where $a_i, b_i \in \mathbb{R}^n \ \forall \ i$. B is initialized with the first l rows of A, so that $B^{(0)} = [a_1, \ldots, a_l]^T$.

The FD algorithm consists in receiving the streams of the various a_i, and updating the last row of B i.e. b_l with the incoming a_i-row. The singular value decomposition of B is then taken, and the singular value of lowest significance is subtracted from every diagonal of the singular value matrix Σ, giving us $S' = \Sigma - \sigma_l$. This means that the last row in S' will be a zero-row. We then update B to be the product of S' and the right singular vector V^T. The last row of B will therefore always be zero, giving space for the next incoming row a_{i+1} in the stream.

Doing this procedure iteratively will give the most dominant directions, as it is the singular values and thereby directions of lowest value that are always discarded when computing B. Note that B is also always at most of rank $l - 1$, as it continuously has a zero-row.

Algorithm 2 Frequent Directions

Input: large matrix $A \in \mathbb{R}^{n \times m}$, sketch matrix $B \in \mathbb{R}^{l \times m}$ consisting only of zeros

Initialize the first $l-1$ rows of B with the $l-1$ first rows of A, $[a_1, \ldots, a_l]$. ▷ The last row, b_l, will be zero-valued

for $i = l, \ldots, m$ **do**

  $b_l = a_i$ ▷ Setting the last row of B to a_i

  $U\Sigma V^T = svd(B)$ ▷ Recall $\Sigma = diag(\sigma_1, \ldots, \sigma_l)$

  $S' = diag\left(\sqrt{\sigma_1^2 - \sigma_l^2}, \sqrt{\sigma_2^2 - \sigma_l^2}, \ldots, \sqrt{\sigma_{l-1}^2 - \sigma_l^2}, \sqrt{\sigma_l^2 - \sigma_l^2}\right)$

  $B = S'V^T$ ▷ The last row will be zero-valued ($\sqrt{\sigma_l^2 - \sigma_l^2} = 0$)

end for

return B

4 System reduction

Model order reduction of artificial neural networks aims to approximate the function of a full, complex artificial neural network, but with less computation time and space taken up. A prerequisite to conduct model order reduction is thus to have a fully trained model.

The system reduction of the neural network as described in [36] gives us a reduced model consisting of three parts:

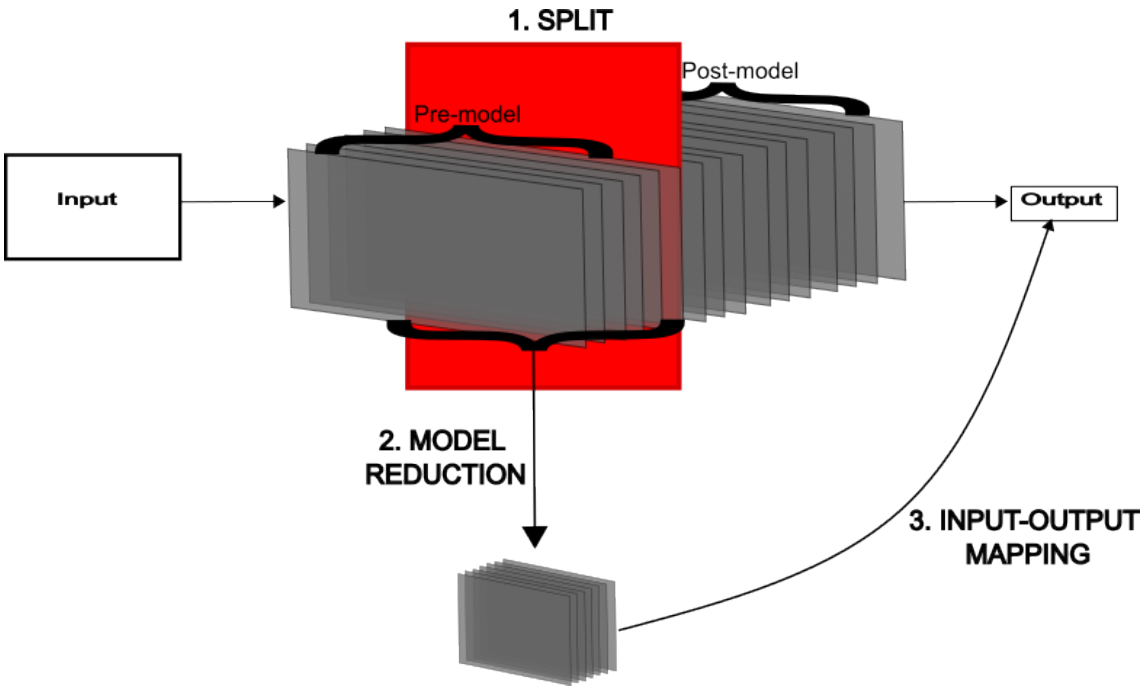


Figure 4.1: A visualization of the architecture of the reduced model in three parts.

1. **Network splitting.** The aim of this step is to process the data through l layers, according to the information amount we are interested in. We therefore split the network into a pre- and post-model, and send data through the pre-model.

2. **Reduction layer.** The aim here is to project the output of the pre-model to a lower dimensional space. This is done by creating a projection matrix. Proposed methods for this are *Active Subspaces* and *Proper Orthogonal Decomposition*.

3. **Input-output mapping.** The aim of this step is to map the reduced data variables to the labels determined by the full neural network. Proposed methods for this are *Polynomial Chaos Expansion* and *Forwardfeeding Neural Network*.

After conducting the model order reduction, it is vital to have a re-learning period for the reduced model, where the parameters of all the three reduced model parts are shifted. This is done using a technique called *knowledge distilliation*. The details regarding the parts of the reduced model as well as the re-learning step will be discussed further.

Algorithm 3 Full Network Reduction Procedure pseudocode

Input:

- The training data set with n_{train} input samples, $\mathcal{D}_{train} = \{x^{(0),i}, y^i\}_{i=0}^{n_t train}$;
- A fully trained artificial neural network, $\mathcal{ANN}(x)$;
- The output of the artificial network, $\{\hat{y^i}\}_{i=0}^{n_{train}} = \{\mathcal{ANN}(x_i)\}_{i=0}^{n_{train}}$;
- The reduced dimension r;
- The index l for the network splitting;

1: $\mathcal{ANN}_{\text{pre}}, \mathcal{ANN}_{\text{post}} = \text{network\_splitting}(\mathcal{ANN}, l)$
2: $x^{(l)} = \mathcal{ANN}_{\text{pre}}(x^{(0)})$
3: $z = \text{reduction\_layer}(x^{(l)}, r)$ ▷ POD or AS
4: $\tilde{y} = \text{input\_output\_map}(z, \hat{y})$ ▷ PCE or FNN
5: Retraining of the reduced neural network

return reduced neural network

4.1 Network splitting

This consists of splitting the fully trained neural network into a pre- and post-model, depending on a chosen cutoff layer l.

A full artificial neural network consisting of L layers is on the form

$$\text{ANN}^L(x^{(0)}) = f_{L+1} \circ f_L \circ \cdots \circ f_1(x^{(0)}) = x^{(L+1)}, \tag{4.1}$$

where $x^{(0)}$ denotes the system input and $x^{(L+1)}$ the system output.

The network splitting consists of choosing the first l layers as the pre-model and the last $L-l$ layers as the post-model. The artificial neural network can therefore be divided into the two following models:

$$\text{ANN}^l_{pre}(x^{(0)}) = f_l \circ f_{l-1} \circ \cdots \circ f_1(x^{(0)}) \tag{4.2}$$

$$\text{ANN}^{L-l}_{post}(x^{(0)}) = f_{L+1} \circ f_L \circ \cdots \circ f_{l+1}(x^{(0)}), \tag{4.3}$$

with the property

$$\text{ANN}^L(x^{(0)}) = \text{ANN}^{L-l}_{post} \circ \text{ANN}^l_{pre}(x^{(0)}), \tag{4.4}$$

meaning the architecture of the full model will be equivalent to the architecture of the composition of the post- and pre-model.

The output from the pre-model will be denoted as $x^{(l)} = \text{ANN}^l_{pre}(x^{(0)})$. Note that the weights and biases from the full model are to be completely restored to the layers of the reduced model.

4.2 Reduction layer

It is important to note that by sending the training set $\{x^{(0),i}\}_{i=1}^{N_{train}}$ into the pre-model, we gain the set of pre-model outputs $\{x^{(l),i}\}_{i=1}^{N_{train}}$. The pre-model outputs is thus the different nodal values

of the l'th layer, for each input in the training set.

Our aim is to project the pre-model output $x^{(l)}$ into a space of lower dimension n. Both the methods described in this subsection consist of trying to find a projection matrix W_{proj}, in order to obtain the reduced solution

$$z = W_{proj}^T x^{(l)}. \tag{4.5}$$

4.2.1 Active Subspaces (AS)

Active Subspaces method [53, 45, 12, 21, 36, 37] is a method basing itself on important directions within the vector space. It is not completely data-driven, but instead based on on probability distributions and relationships between spatial gradients.

We are given an input vector $\boldsymbol{\mu} = [\mu_1, \dots, \mu_n]^T$ with a related probability distribution function $\rho(\boldsymbol{\mu})$. We introduce a so-called function of interest[36][p. 82] $g : \mathbb{R}^n \to \mathbb{R}$. The Active Subspaces method is based on the use of the gradient ∇g to find the low-dimensional subspace of the input where g varies the most on average.

As g is, in this case, defined as a scalar quantity whilst $\mu \in \mathbb{R}^n$, we have that $\nabla g(\mu) \in \mathbb{R}^n$, thus denoting the spatial derivative of $g(\mu)$ with regards to every element in μ. The computation of the derivative can be done either numerically or analytically.

We use the outer product of the average of the gradient with itself to construct an empirical covariance matrix,

$$C = \mathbb{E}[\nabla g(\mu) \nabla g(\mu)^T] = \int \nabla g(\mu) \nabla g(\mu)^T \rho d\mu, \tag{4.6}$$

where $\mathbb{E}$ denotes the expected value.

The outer product of $\nabla g(\mu)$ with itself leads to C being positive definite and symmetric. Such matrices yield solely positive eigenvalues.

It is thereby possible to use eigenvalue decomposition on C, so that we have

$$\mathbf{C} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^T, \quad \boldsymbol{\Lambda} = diag(\lambda_1, \dots, \lambda_n), \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n. \tag{4.7}$$

As stated above, $\boldsymbol{\Lambda}$ is a matrix with the diagonal consisting of eigenvalues in descending order. The columns of $\mathbf{V}$ are the respective eigenvectors $[v_1, \dots, v_n]$. The eigenvectors are thus in descending order according to the importance of them as directions.

Choosing an integer $n_{as} < n$. We decompose V by constructing a matrix V_1 that is composed of the n_{as} first columns, and a matrix V_2 that is composed of the $n - n_{as}$ columns. $\boldsymbol{\Lambda}$ is equivalently decomposed into a matrix $\Lambda_1 \in \mathbb{R}^{n_{as} \times n_{as}}$ and a matrix $\Lambda_2 \in \mathbb{R}^{n \times n}$. We thereby have

$$\mathbf{V} = \begin{bmatrix} V_1 & V_2 \end{bmatrix}, \quad \boldsymbol{\Lambda} = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}, \quad V_1 \in \mathbb{R}^{n \times n_{as}}, \quad V_2 \in \mathbb{R}^{n \times n - n_{as}}. \tag{4.8}$$

V_1 denotes the range of the *active subspace* and V_2 that of the *inactive subspace*. We aim to project the input μ onto the active subspace.

The orthogonality of the eigenvector basis gives us the property $\mathbf{V}\mathbf{V}^T = I_n$, where I_n denotes the identity matrix of dimension n.

We can thereby rewrite our input as

$$\boldsymbol{\mu} = \mathbf{V}\mathbf{V}^T\boldsymbol{\mu} = [V_1 \quad V_2][V_1 \quad V_2]^T\boldsymbol{\mu} = V_1V_1^T\boldsymbol{\mu} + V_2V_2^T\boldsymbol{\mu}. \tag{4.9}$$

From the above equations we define the reduced variables $\tilde{\mu_1} = V_1^T\boldsymbol{\mu} \in \mathbb{R}^{n_{AS}}$ and $\tilde{\mu}_2 = V_2^T\boldsymbol{\mu} \in \mathbb{R}^{n - n_{AS}}$, respectively denoted as the *active variable* and *inactive variable*. This effectively reduces our equation to

$$\boldsymbol{\mu} = V_1\tilde{\mu}_1 + V_2\tilde{\mu}_2, \tag{4.10}$$

where our input μ is thus expressed as the sum of the active and inactive subspace multiplied with an active and inactive variable.

From this we can find the approximation

$$\boldsymbol{\mu} \approx V_1 \tilde{\mu}_1, \tag{4.11}$$

where the inactive variable is discarded due to insignificance.

Using V_1 as a projection matrix thus gives a reduced variable where the directions of the active subspace is taken into consideration.

4.2.2 Proper Orthogonal Decomposition (POD)

Proper Orthogonal Decomposition [22, 37, 36] is a data-driven method for constructing a reduced order model. It relies solely on the data points of the input vector to construct a reduced order basis. POD is used in many different areas of study, for example in the solution of differential equations [4]. The same terminology will therefore be used here.

We denote $S = [u_1, \ldots, u_{n_s}]$ as a matrix where each column contains an input vector u_i, and n_s denotes the amount of samples within the data set. As we have that the input vectors $u_i \in \mathbb{R}^n$, we must have that $S \in \mathbb{R}^{n \times n_s}$. Our matrix S is often referred as the Snapshot matrix, as it may contain the snapshots of various solutions.

Using Singular Value Decomposition as described in 3.2, we decompose S so that

$$S = U \Sigma V^H. \tag{4.12}$$

Σ contains singular values in descending order, U consists of the respective singular left vectors, and V^H the respective singular right vectors.

For the sake of notation, we denote the columns of U as modes. Our aim here is to project our matrix S onto a low dimensional space spanned by its modes. Choosing an integer $n_{POD} < n$, we can truncate U by discarding the $n - n_{POD}$ last modes, so that we have

$$U \approx U_{n_{POD}} \in \mathbb{R}^{n \times n_{POD}}, \tag{4.13}$$

where the modes that are retained in the reduced matrix are the n_{POD} first modes. We've thereby discarded the less significant modes, as they belong to the smaller singular values due to the descending order.

Using our truncated matrix $U_{n_{POD}}$ as a projection matrix gives us the reduced variable

$$S^{POD} = U_{n_{POD}}^T S \in \mathbb{R}^{n_{POD} \times n_s}, \tag{4.14}$$

which now is spanned within the reduced subspace of the most significant nodes.

4.3 Input-output mapping

Here we need a mapping to correlate z to $\hat{y}$ (the final output of the full model). Note that we are *not* correlating them to their actual labels, but to the labels predicted by the full neural network. This is done with the aim of having the input-output mapping approximate the *full model*; not to generate a new model altogether.

4.3.1 Polynomial Chaos Expansion (PCE)

The theory of Polynomial Chaos Expansion [25] states that the a model output $X : \mathbb{R}^R \to \mathbb{R}$ can be expanded as a sum of orthogonal multivariate polynomials, so that

$$X(\boldsymbol{\xi}) = \sum_{i=0}^{\infty} c_i \phi_i(\boldsymbol{\xi}), \tag{4.15}$$

where $\boldsymbol{\xi}$ denotes the input vector $\boldsymbol{\xi} = [\xi_1, \xi_2, \ldots, \xi_R]$. For each orthogonal multivariate polynomial $\phi_i(x)$, we have an associated unknown coefficient c_i representing the weight of the polynomial.

Note that the input vector $\boldsymbol{\xi}$ is assumed to consist of independent, random variables that have an associated probability density function $\rho(\xi_i)$. The basis functions $\Phi(\xi_i)$ will thus be orthonormal to one another with respect to the density function.

The finite representation of this can be found by truncating at the $(P+1)$'th term, so that we have an approximation

$$X(\boldsymbol{\xi}) \approx \sum_{i=0}^{P} c_i \phi_i(\boldsymbol{\xi}). \tag{4.16}$$

Recall that R denotes the dimension of the input vector ξ. Let p denote the highest order of the multivariate polynomials in the dimension R, so that given any index i, we have that $\phi_i(x) \in \mathbb{R}^p$.

The term of truncation is given, by the combination formula, as $P + 1 = \frac{(p+R)!}{p!R!}$. The number of unknown coefficients c_i will thereby also be $P + 1$.

The assumption that $\xi_1, \ldots, \xi_R$ are independent parameters allows us to rewrite the multivariate $\phi_i(x)$ as the product of one-dimensional functions:

$$\boldsymbol{\phi}_i(\boldsymbol{\xi}) = \boldsymbol{\phi}_i(\xi_1, \ldots, \xi_R) = \prod_{k=1}^{R} \phi_k^{d_k}(\xi_k), \quad i = 0, \ldots, P, \quad d_k = 0, \ldots, p, \quad \sum_{k=1}^{R} d_k \leq p. \tag{4.17}$$

What this essentially means is that each multivariate function $\boldsymbol{\phi}_i(\boldsymbol{\xi})$ can be rewritten as the product of one-dimensional functions belonging to each parameter of the input vector, $\phi_i(\xi_i)$. The order d_k of the one-dimensional functions $\phi_i(x)$ can be anywhere in the domain $[0, \ldots, p]$, so as long as the order of the multivariate function $\boldsymbol{\phi}_i(x)$ doesn't surpass p, meaning that the restriction $\sum_{K=1}^{R} d_k \leq p$ needs to be held.

What is left is to determine the one-dimensional orthogonal functions $\phi_i(x)$ and the coefficients of the multivariate functions c_i. $X(\boldsymbol{\xi})$ consists of independent random variables, and so they must follow a probability distribution of some sort. From [28] we can set the orthogonal polynomials as those belonging to the respective distribution. For example, assuming a Gaussian distribution would lead us to use the Hermite polynomials as univariate polynomials $\phi_i(x)$.

In order to estimate the coefficients c_i we use a regression method, as described in section 3.1. The following minimization problem will be solved:

$$\mathbf{c} = \arg\min_{c^* \in \mathbb{R}^P} \frac{1}{n_{input}} \sum_{j=1}^{n_{input}} \left(\hat{X} - \sum_{j=0}^{p} c_{,}^* \phi_{,}(\xi^i) \right), \tag{4.18}$$

where n_{input} denotes the amount of input samples ξ^j used for the optimization problem. The asterics j of ξ^j thus denotes the sample number of the respective input.

Solving this as a least squares problem, we define the following matrix Φ which will be analogous to our matrix A from equation (3.4):

$$\Phi = \begin{bmatrix} \phi_1(\xi^1) & \phi_2(\xi^1) & \ldots & \phi_P(\xi^1) \\ \phi_1(\xi^2) & \phi_2(\xi^2) & \ldots & \phi_P(\xi^2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\xi^{n_{input}}) & \phi_2(\xi^{n_{input}}) & \ldots & \phi_P(\xi^{n_{input}}) \end{bmatrix}. \tag{4.19}$$

Each row of Φ thus denotes the polynomial chaos expansion of an arbitrary sample ξ^j of the input, with each column belonging to one of the orthogonal functions ϕ. This means that the amount of columns is equivalent to the amount of unknown coefficients, $P+1$.

Example 4.1. *Setting $p = 0$. Assuming that X has a Gaussian distribution. Our amount of columns in Φ would be $P + 1 = \frac{(p+R)!}{p!R!} = \frac{(R)!}{R!} = 1$.*

The only Hermite polynomial of an order that is equivalent or less to p would be $H_0(x) = 1$, and so our matrix Φ would be on the following form:

$$\Phi = \begin{bmatrix} H_0(\xi^1) \\ H_0(\xi^2) \\ \vdots \\ H_0(\xi^{n_{input}}) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{4.20}$$

Example 4.2. *Setting $p = 1$. Assuming that X has a Gaussian distribution. The amount of columns in Φ would be $P + 1 = \frac{(1+R)!}{1!R!} = \frac{(1+R)!}{R!} = (1 + R)$.*

The only relevant Hermite polynomials would be $H_0(x) = 1$ and $H_1(x) = 2x$. From [refer to equation about the product of equations here] we have the property that $\sum_{k=0}^{R} d_k \leq p$, meaning that any $H_1(\xi_i)$ can only be multiplied with $H_0(\xi_j) = 1$ for the remainder of the terms. Thus, we will have one column denoting the 1-vector from the $p = 0$ basis, and R columns denoting $H_1(\xi_i)$, where $i = 1, \ldots, R$. Our matrix Φ would thus be on the following form:

$$\Phi = \begin{bmatrix} H_0(\xi^1) & H_1(\xi_1^1) & \ldots & H_1(\xi_R^1) \\ H_0(\xi^2) & H_1(\xi_1^2) & \ldots & H_1(\xi_R^2) \\ \vdots & \vdots & \ddots & \vdots \\ H_0(\xi^{n_{input}}) & H_1(\xi_1^{n_{input}}) & \ldots & H_1(\xi_R^{n_{input}}) \end{bmatrix} = \begin{bmatrix} 1 & 2\xi_1^1 & \ldots & 2\xi_R^1 \\ 1 & 2\xi_1^2 & \ldots & 2\xi_R^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2\xi_1^{n_{input}} & \ldots & 2\xi_R^{n_{input}} \end{bmatrix}. \tag{4.21}$$

Example 4.3. *Setting $p = 2$. Assuming that X has a Gaussian distribution. The amount of columns in Φ would be $P + 1 = \frac{(2+R)!}{2!R!} = \frac{(R+1)(R+2)}{2}$.*

The relevant Hermite polynomials would be $H_0(x) = 1$, $H_1(x) = 2x$ and $H_2(x) = 4x^2 - 2$. Again due to $\sum_{k=0}^{R} d_k \leq 2$, we have a larger yet still limited amount of combinations available. From the $p = 1$ basis we have the columns of $H_0(\xi^i)$. Furthermore, there will be R columns of $H_1(\xi^j)$, where $j = 1, \ldots, R$.

Our second order Hermite polynomial can only be combined with polynomials of order 0, so that there must be R columns of $H_2(\xi^j)$ as well.

The only combination of polynomials (not including 0'th order polynomials) will be Hermite polynomials of first order. It is only possible to combine two Hermite polynomials of first order with one another without surpassing the limit of the order. We thereby have $P + 1$ columns of

$$H_1(\xi_i) \cdot H_1(\xi_j), \quad i, j = 0, \ldots, R, \quad i \neq j, \tag{4.22}$$

where the commutativity of multiplication should make it obvious that all columns will be unique.

Thus, Φ will be on the following form:

$$\Phi = \begin{bmatrix} 1 & H_1(\xi_1^1) & \ldots & H_1(\xi_R^1) & H_1(\xi_1^1) \cdot H_1(\xi_2^1) & \ldots & H_1(\xi_{R-1}^1) \cdot H_1(\xi_R^1) & H_2(\xi_1^1) & \ldots & H_2(\xi_R^1) \\ 1 & H_1(\xi_1^2) & \ldots & H_1(\xi_R^2) & H_1(\xi_1^2) \cdot H_1(\xi_2^2) & \ldots & H_1(\xi_{R-1}^2) \cdot H_1(\xi_R^2) & H_2(\xi_1^2) & \ldots & H_2(\xi_R^2) \\ \vdots & \vdots & \ldots & \vdots & \vdots & \ldots & \vdots & \vdots & \ldots & \vdots \\ 1 & H_1(\xi_1^{n_{in}}) & \ldots & H_1(\xi_R^{n_{in}}) & H_1(\xi_1^{n_{in}}) \cdot H_1(\xi_2^{n_{in}}) & \ldots & H_1(\xi_{R-1}^{n_{in}}) \cdot H_1(\xi_R^{n_{in}}) & H_2(\xi_1^{n_{in}}) & \ldots & H_2(\xi_R^{n_{in}}) \end{bmatrix}. \tag{4.23}$$

We use the method presented in section 3.1 to compute the coefficients c:

$$\mathbf{c} = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \hat{X}, \tag{4.24}$$

giving us the most optimal solution to the minimization problem presented.

4.3.2 Forward-feeding Neural Network (FNN)

In order to have an input-output function, we can use a form of function approximation of which this thesis is based on, namely using an artificial neural network.

Note that we utilize a fully connected forward-feeding neural network here. As stated in section 2.4.2, feed-forward neural networks are ANNs where every connection between layers is in the same direction; from input to output direction. Fully connected means that each respective node in a layer i points to every node in the next layer $i+1$.

Defining an FNN consisting of L hidden layers, with an output from the network given as $\hat{y}$. We have from section 2.4.2 that the output from a FNN is given as

$$\begin{aligned}\hat{y_j}^{\mathcal{FNN}} = \sigma(z^{(L)}) = \sigma\left(\sum_{i=0}^{n_L} w_{ij}^{(L)} x_i^{(L-1)} + b_j^{(L)}\right) = \sigma\left(\sum_{i=0}^{n_L} w_{ji}^{(L)} \sigma\left(\sum_{q=0}^{n_{L-1}} w_{iq}^{(L-1)} x_q^{(L-2)} + b_i^{(L-1)}\right) + b_j^{(L)}\right) \\ = \cdots = \sigma\left(\sum_{i=0}^{n_L} w_{ij}^{(L)} \sigma\left(\sum_{q=0}^{n_{L-1}} w_{jq}^{(L-1)} \sigma\left(\ldots\left(\sigma\left(\sum_{p=0}^{n_1} w_{sp}^{(1)} x_p^{(0)} + b_s^{(1)}\right) + b_r^{(2)}\right)\ldots\right) + b_j^{(L-1)}\right) + b_i^{(L)}\right).\end{aligned} \tag{4.25}$$

Note that the various n_i denote the amount of neurons at the respective layer i.

4.4 Re-Training

After constructing the reduced neural network, it is vital to re-train the model to improve the accuracy. The re-training is done using *Knowledge Distillation* methods, following the procedure of [13, 37].

The thought behind Knowledge Distillation methods (KD) is to distill the knowledge from a large model to a smaller model through a training process[20, 23]. We train our smaller model using the soft labels from the large model[6]. The soft labels could be the activation from the softmax- or softplus-function, giving class distributions, with a *temperature factor* to decrease the certainty of the predictions.

We refer to the full model as the *teacher model*, whilst the smaller model will be referred to as the *student model*. This is intact with the name of the method, Knowledge Distillation, as it is the student model that will "learn" from the teacher model.

We denote $\hat{y}$ as the output of the last layer in a deep neural network. The probability that an input belongs to a certain class i is given by the softmax function as introduced in section 2.4.5,

$$p_i = \frac{\exp(\hat{y}_i)}{\sum_{j=0}^{n_{class}} \exp(\hat{y}_j)}. \tag{4.26}$$

We introduce a temperature factor T here, used to control the importance of each target[57]. The softmax function with a temperature factor is given as

$$p_i = \frac{\exp(\frac{\hat{y}_i}{T})}{\sum_{j=0}^{n_{class}} \exp(\frac{\hat{y}_j}{T})}, \tag{4.27}$$

where the temperature "softens" the output $\hat{y}$ within the loss function i.e. makes its predictions less certain. The aim of the temperature factor is to avoid high penalizations of wrong results from the model. Note that, for the purpose of this thesis, it is assumed that the temperature factor T is always greater than 1. The cases where T is equivalent to 1 are equivalent to using no temperature factor at all.

The *distillation loss* refers to the loss between the output of the teacher model and the student model. It is given as

$$\mathcal{L}_D(p(\hat{y}_t, T), p(\hat{y}_s, T)) = \mathcal{L}_{KL}(p(\hat{y}_t, T), p(\hat{y}_s, T)), \tag{4.28}$$

where $\mathcal{L}_{KL}$ denotes the Kullback-Leibler Divergence Loss function, as described in section 2.3.1. Note that $\hat{y}_s$ and $\hat{y}_t$ denote the output of the student model and the teacher model respectively. Recall that this loss function based itself on the probability distributions of the model output and the desired output, given in the case of a temperature factor as

$$\mathcal{L}_{KL}(p(\hat{y}_t, T), p(\hat{y}_s, t)) = T^2 \sum_{i=0}^{N} p_i(\hat{y}_{t,i}, T) \log\left(\frac{p_i(\hat{y}_{t,i}, T)}{p_i(\hat{y}_{s,i}, T)}\right), \tag{4.29}$$

where $p(x)$ is the probability distribution of the teacher model and $q(x)$ is the probability distribution of the student model, in the case of knowledge distillation.

The student loss is given as

$$\mathcal{L}_S(\hat{y}, p(\hat{y}_s, T)) = \mathcal{L}_{CE}(y, p(\hat{y}_s, T)), \tag{4.30}$$

where $\mathcal{L}_{CE}$ denotes the cross-entropy loss as provided in section 2.3.1. Recall that y from the cross-entropy was not an output, but instead the binary truth value of a model. From section 2.3.1 we have the cross-entropy loss function given as

$$\mathcal{L}_{CE}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} \left[y_j \ln \hat{y}_j + (1 - y_j) \ln(1 - \hat{y}_j) \right]. \tag{4.31}$$

Knowledge distillation is a response-based method that transfers the knowledge from the teacher model to the student model by mimicking the final prediction of the teacher model. The final loss used in the training process is therefore the weighted sum between the distillation loss and the student loss, given as

$$\mathcal{L}(x^{(0)}, w) = \lambda \mathcal{L}_D(p(\hat{y}_t, T), p(\hat{y_s}, T)) + (1 - \lambda)\mathcal{L}_S(y, p(\hat{y}_s, T = 1)), \tag{4.32}$$

where $x^{(0)}$ is the input to the model and w the weights of the student model. The parameters λ acts a regularization parameter, affecting the importance of either the student model or the teacher model in the equation. A larger λ giver lower significance to the output of the student model, and more significance to the difference between the teacher and student model.

5 Implementation

All numerical implementations were done in Python, using the package *torchvision* for loading the data sets and constructing the full neural network. Torchvision is a part of the *PyTorch*[42] project, which is an open-sourced machine learning framework. The rest of the implementations were done manually, with the exception of Active Subspaces for which I imported the *ATHENA* library[11].

The code that has been implemented can be found in the github repository [32]. It is important to keep in mind that the implementation of the re-training using knowledge distillation is taken from [8, 34].

5.1 Data sets

Two different data sets (CIFAR-10 and SVHN) were utilized for the purpose of this thesis. Both consist of 32×32 color images i.e. with 3 channels and thus the input size $(32, 32, 3)$. This means that the images are not of very high resolution [59], which can cause difficulties in the task of classifying them. The model we used has a pre-determined input size of $(3, 224, 224)$. A transformation to this size was tried on both data sets, however this led to large challenges regarding memory on both local and external servers. Instead, we opted for a modification of the input size of the neural network.

Both data sets consist of 10 labeled classes, with a difference in the data imbalance. Although image classification will be implemented on both data sets, the subclass of image classification varies: CIFAR-10 deals with object recognition, whilst SVHN deals with digit recognition. Due to the large differences in what the data sets represent, there will be two fully trained models accommodating directly to the data sets: $\mathcal{ANN}_{\text{CIFAR}}$ and $\mathcal{ANN}_{\text{SVHN}}$. There will also be two models for each respective data set that have been trained on augmented data, in order to see the impact of the robustness of the networks on the reduced models. The models trained on augmented data will be denoted $\mathcal{ANN}_{\text{CIFAR-augm}}$ and $\mathcal{ANN}_{\text{SVHN-augm}}$.

5.1.1 CIFAR-10

The CIFAR-10 data set[29, 9] is a data set consisting of 60000 pictures with belonging labels: 50000 are in the training set, whilst 10000 are in the testing set.

There are 10 classes altogether, comprised of different vehicles and animals. The data set is perfectly well-balanced: each of the 10 classes has exactly 6000 images with labels represented in the set, where 5000 are in the training set and 1000 in the testing set.

Figure 5.1 shows some samples from each class of the original CIFAR-10 images.

The augmentation consisted of normalizing the color scale according to the standard deviation and mean of CIFAR-10, random horizontal flips, and random crops of images with a padding of 4.

5.1.2 Street House View Numbers (SVHN)

The Street House View Numbers data set [38, 54] consists of 99289 images of house numbers from a street view, obtained from Google Street View images. It therefore closely resembles the famous data set with handwritten letters, MNIST[15], with the difference that it is with colors and with a perhaps more "blurred" view of the various numbers due to the pictures being taken from the streets. In cases where a house has a multi-digit number, the image is cropped so that the label of the image is given is at the centre of the image.

In figure 5.3 one can see some samples from the SVHN data set. As described, at times the images are cropped due to multiples digits in the house number.

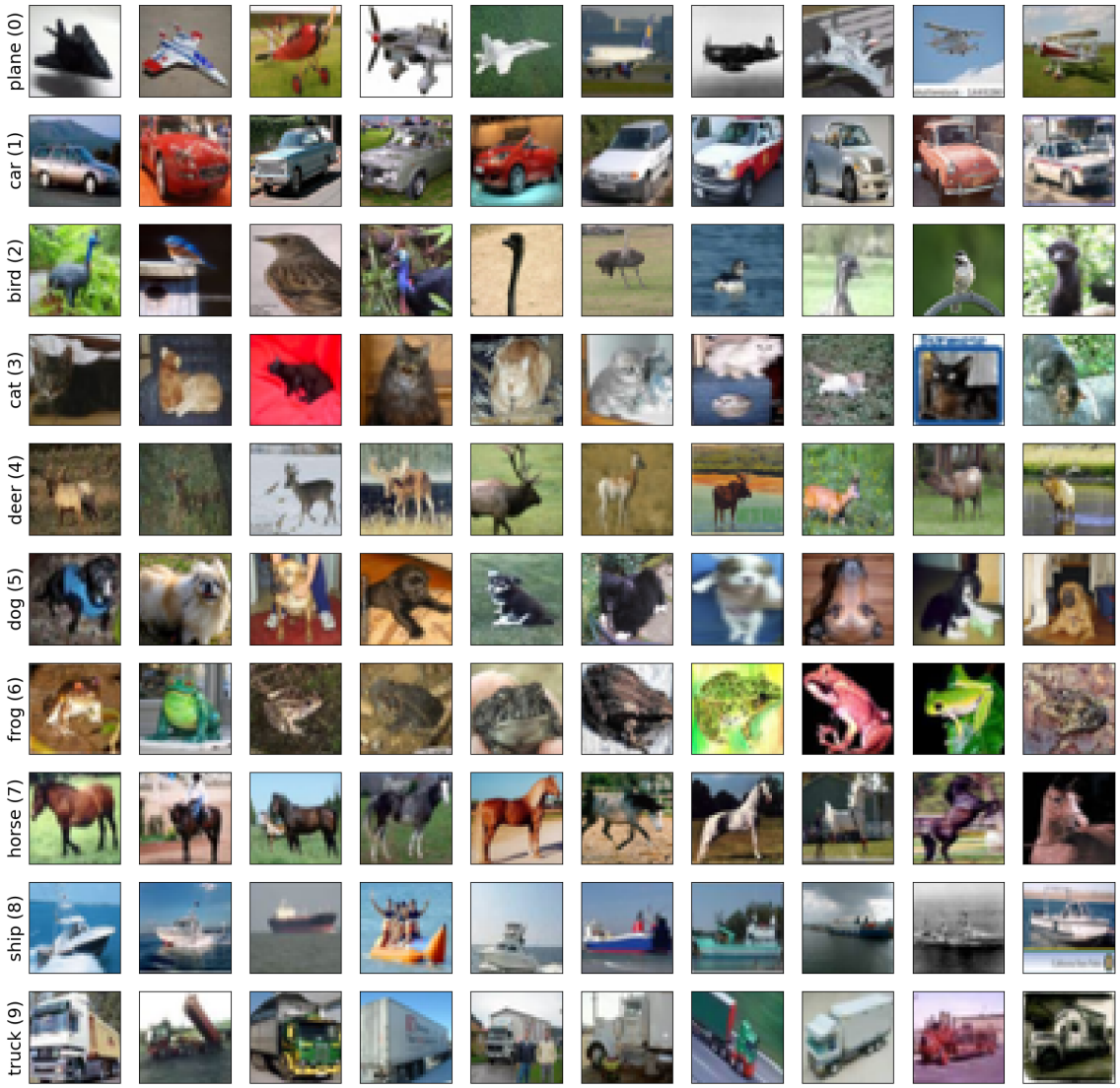


Figure 5.1: 10 randomly selected images from each class of the CIFAR-10 data set. The class of each row is captioned to the left, with their respective label.

Figure 5.4 shows the same images after data augmentation and normalization using statistics from the data set. Many digits are not readable to the human eye after this due to small color distinctions in different shots.

The amounts and percentages of each class represented in the dataset is given in table 1. We can see that this is a dataset that is not particularly well-balanced, with percentages in the training set ranging from 6.36% to 18.92%. This may lead to challenges for the classification model, as it will in most cases

5.2 CNN model

In order to replicate the procedure from [36], we have constructed the VGG-16 neural network [51]. VGG stands for "Visual Geometry Group", and is a convolutional neural network group presented by Oxford University that won the ImageNet competition in 2015, performing with a 92.7% top-5 accuracy. There are several models in the VGG-group, all named VGG-xx, where xx denotes the amount of layers in the respective network. Note that this does not take into account pooling layers.

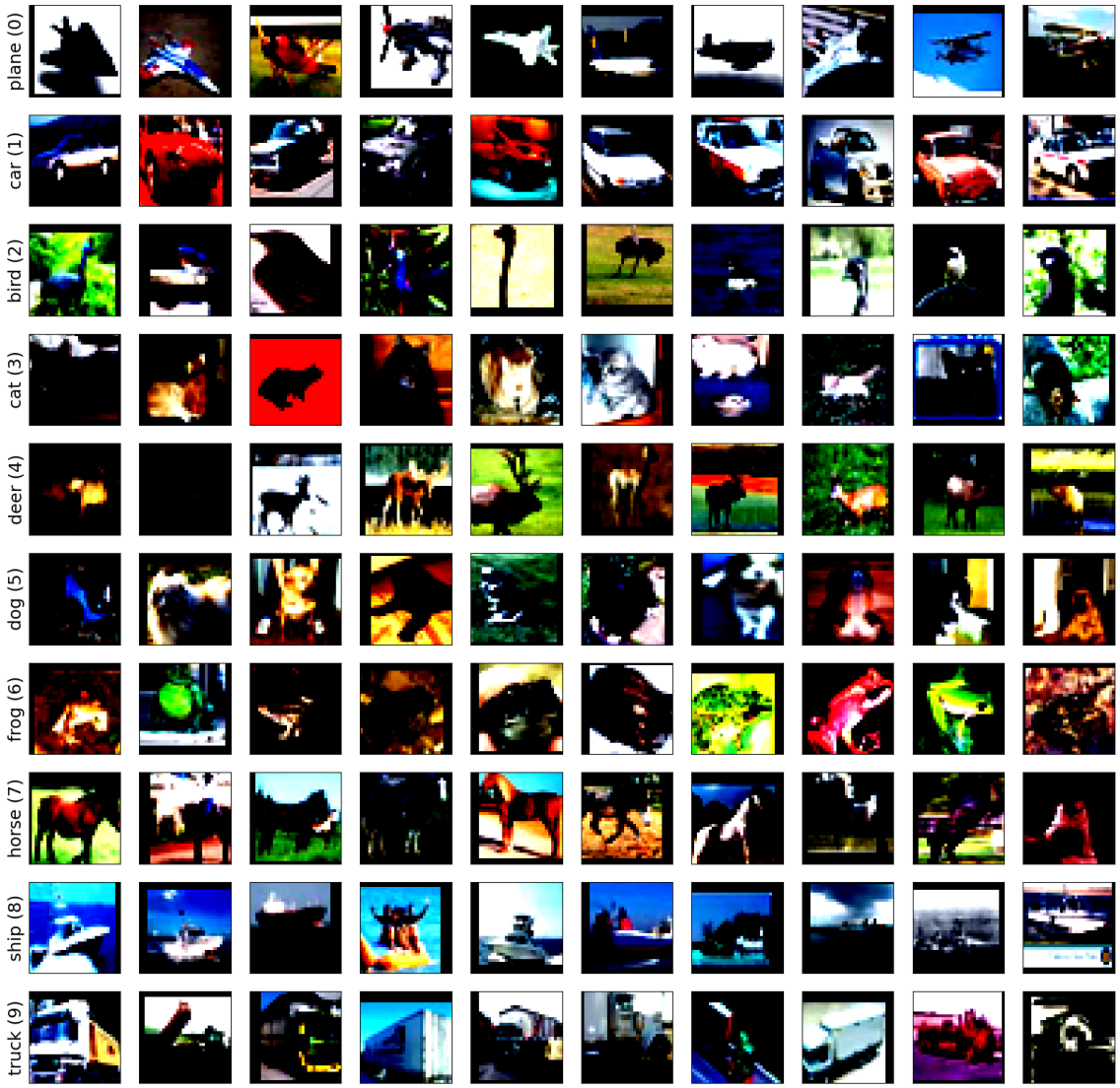


Figure 5.2: The same 10 images from each class of CIFAR-10 as in figure 5.1, but with normalization and random data augmentation.

Digit	**Training set**		**Testing set**		**Full data set**	
	Amount	*Percentage*	*Amount*	*Percentage*	*Amount*	*Percentage*
0	4948	6.75%	1744	6.7%	6692	6.74%
1	13861	18.92%	5099	19.59%	18960	19.1%
2	10585	14.45%	4149	15.94%	14734	14.84%
3	8497	11.6%	2882	11.07%	11379	11.46%
4	7458	10.18%	2523	9.69%	9981	10.05%
5	6882	9.39%	2384	9.16%	9266	9.33%
6	5727	7.82%	1977	7.59%	7704	7.76%
7	5595	7.64%	2019	7.76%	7614	7.67%
8	5045	6.89%	1660	6.38%	6705	6.75%
9	4659	6.36%	1595	6.13%	6254	6.3%
Total	73257	100%	26032	100%	99289	100%

Table 1: The representation of the different digits within the SVHN dataset.

The VGG-group is based on (3×3) convolutions, however each layer consists of a lot of filters. Due to the large amount of filters, the entire VGG-16 model has 138 million parameters that

Figure 5.3: 10 randomly selected images from each class of the SVHN data set. The class of each row is captioned to the left.

need tuning. This results in the training process, as well as any forward pass in general, being computationally expensive and time-consuming. This motivates the usage of model order reduction on the VGG-16 model.

The VGG-16 model consists of 13 convolutional layers with 5 max-pooling layers well-spread between convolutional blocks. After the 13 convolutional + 5 max-pooling layers, there are three fully connected layers, followed by the softmax activation function. The final three layers in the architecture, together comprising a FNN as decribed in 2.1, force the model to commit classification. The final softmax function gives a probabilistic score of the classes each input image belongs to, meaning the predicted label is found as the index of the largest value gathered from the softmax function.

Figure 5.5 depicts the general architecture of a VGG-16 model, with the original amount of parameters being present.

Some important specifications to the model architecture, in case of manual re-production of the model:

- Each convolutional layer consists of (3×3) filters with stride=1 and padding=1.

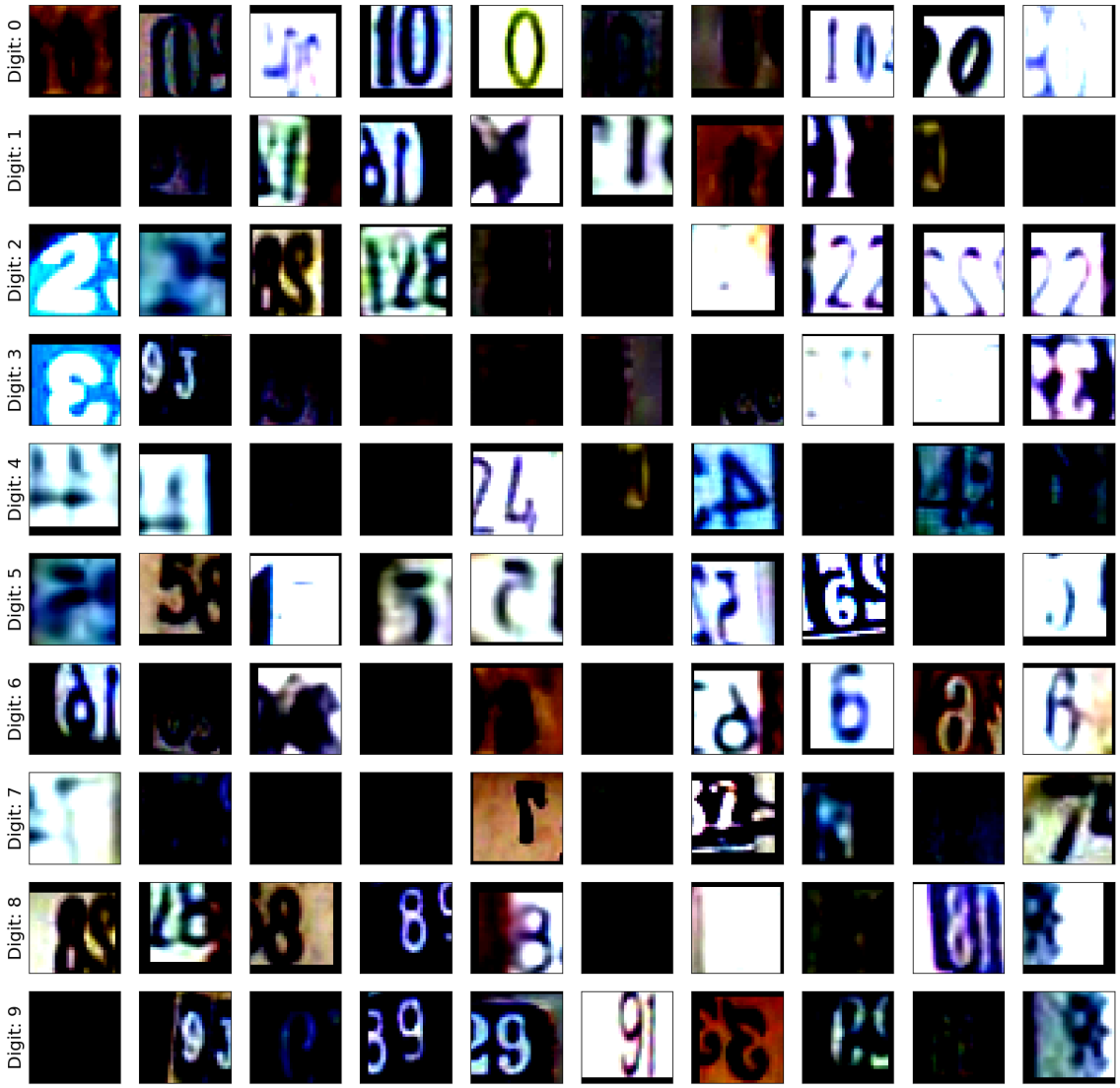


Figure 5.4: 10 randomly selected images from each class of the SVHN data set. The class of each row is captioned to the left.

- The activation function implemented after each convolutional layer is ReLU. This is not visualized in 5.5, as it is already specified in section 2 that this is the structure of a convolutional layer.
- Each max-pooling layer is of size 2 with stride=2.

In order to accommodate for a classification problem with 10 labels featuring images of 32×32 pixels, the architecture of VGG-16 was modified to the architecture of figure 5.6. This especially affects the outputs of the various layers, which is viable in the case of data-driven model reduction.

It is important to note that, for the scope of this thesis, the layers will not be enumerated according to the amount of convolutional layers as in figure 5.6. When decomposing the VGG-16 model from torchvision sequentially, the activation layers ReLU are taken into consideration as a full layer. When referring to indices of layers and so on, a convolutional layer will therefore take up the space of two indices, where one index will consist of the filters and the other of the activation function that follows.

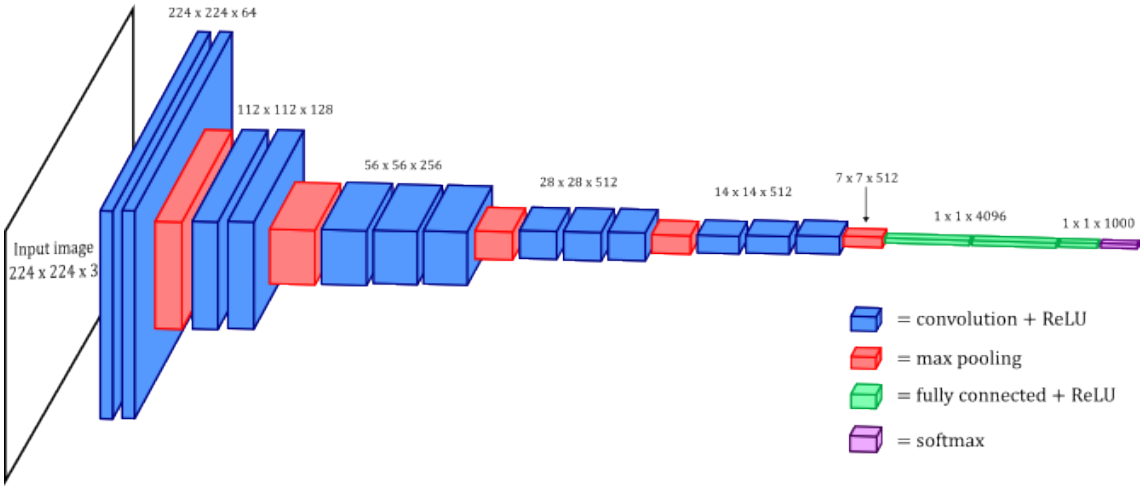


Figure 5.5: General architecture of the convolutional network VGG-16, with an input image consisting of 224×224 pixels, for a classification problem with 1000 categories

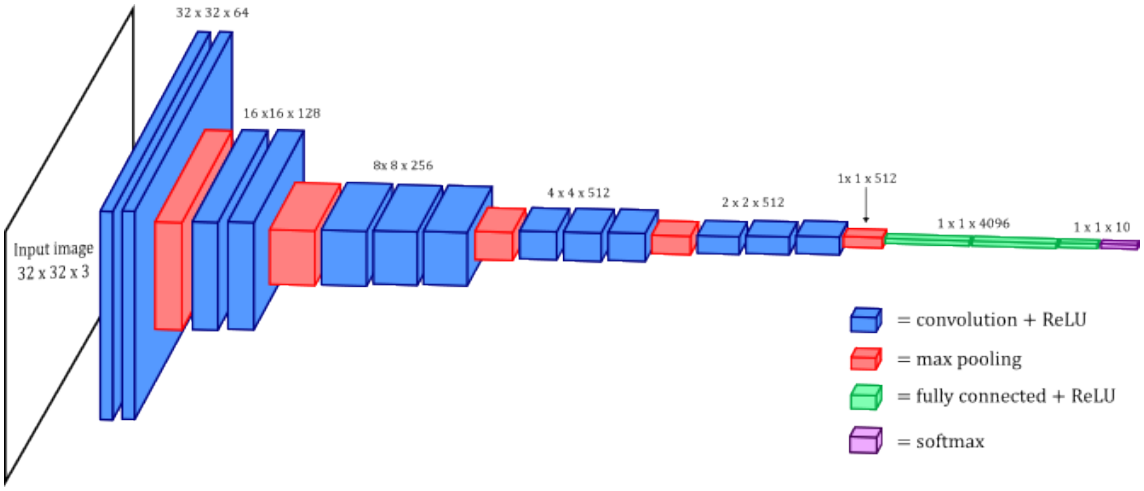


Figure 5.6: Modifications to the VGG-16 model for input images of 32×32 pixels, belonging to a classification problem consisting of 10 labels.

5.3 Specifications to methods

Section 4 introduced various numerical methods for system reduction. The specifications to these according to our model and data sets will be presented here.

5.3.1 Network splitting

As described in section 4, a layer-index l is chosen to be where we want to split our full model into a pre-model and a post-model:

$$\mathcal{ANN}^{l}_{pre}(x^{(0)}) = f_{l-1} \circ f_{l-1} \circ \cdots \circ f_0(x^{(0)}) \tag{5.1}$$

$$\mathcal{ANN}^{L-l}_{post}(x^{(0)}) = f_L \circ f_{L-1} \circ \cdots \circ f_l(x^{(0)}), \tag{5.2}$$

The parameters of the full model are kept intact, so that

$$\mathcal{ANN}^{L}(x^{(0)}) = \mathcal{ANN}^{L-l}_{post} \circ \mathcal{ANN}^{l}_{pre}(x^{(0)}).$$

As the convolutional layers consist of the convolutional filters combined with an activation function layer, it is viable not to separate these. Therefore, the parameter l will not be chosen to be any index separating a convolutional layer. From [36] and [8] we see that the indices chosen are not directly in contact with a pooling layer. The pooling layer does not have any parameters such as weights and biases, which is why model reduction with the aim of reducing parameters does not deem pooling layers as candidates for the split. On the other hand, performing network splitting at a layer l following a max-pooling layer, we reduce the dimension of the pre-model output whilst retaining the most important features.

5.3.2 Model reduction layer

From our reduced model $\mathcal{ANN}_{pre}$, we compute the pre-model output

$$x^{(l)} = \mathcal{ANN}^{l}_{pre}(x^{(0)}). \tag{5.3}$$

The pre-model output is the variable that we want to reduce. This is done through constructing a projection matrix $W_{proj} \in \mathbb{R}^{n_l \times r}$, where r is the lower dimension we want to project the pre-model output onto. This gives us the reduced output

$$z = W^{T}_{proj} x^{(l)}. \tag{5.4}$$

We implement either Active Subspaces or Proper Orthogonal Decomposition to construct the projection matrix. They have been introduced more in-depth in section 4, however the specifics will be detailed here.

Active Subspaces We can assume that that $x^{(l)}$ is characterized by a probability density function $\rho(\mu)$.

Introducing the function of interest, which in our case is the loss-function of the post-model, so that we have

$$g_l(x^{(l)}) = \text{loss}(\text{ANN}^{l}_{post}(x^{(l)})). \tag{5.5}$$

We want to find the empirical covariance matrix. As all our input and output is discrete, we need the discrete equation for the empirical covariance matrix as well. In our case it is given as

$$\hat{C} = \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} \nabla_l g(x^{i,(l)}) \nabla_l g(x^{i,(l)})^T, \tag{5.6}$$

where i denotes the respective sample in the training set.

Calculating the eigenvalue decomposition can be computationally expensive. Due to this the *Frequent Directions method* has been proposed in [36] for approximating the SVD decomposition. The Frequent Directions method is already implemented in the Python library utilized, ATHENA.

Following the procedure of Frequent Directions method, we introduce

$$\hat{G} = \begin{bmatrix} \nabla_l g(x^{1,(l)}) & \nabla_l g(x^{2,(l)}) & \dots & \nabla_l g(x^{n_{train},(l)}) \end{bmatrix}, \tag{5.7}$$

where each row is a flattened version of the gradient of the r first input values. The SVD is given as $\hat{G} = \hat{U}\hat{\Sigma}\hat{V}^T$. From the fact that $\hat{G}^T\hat{G} = \hat{C}$, we have the singular value decomposition

$$\hat{C} = (\hat{U}\hat{\Sigma}\hat{V}^T)^T\hat{U}\hat{\Sigma}\hat{V}^T = \hat{V}\hat{\Sigma}\hat{U}^T\hat{U}\hat{\Sigma}\hat{V}^T = \hat{V}\hat{\Sigma}\hat{\Sigma}\hat{V}^T, \tag{5.8}$$

where the orthogonality of the singular vectors gives us that $\hat{U}^T\hat{U} = I$. Notice that the term to the right of equation (5.8) is identical to that of an eigenvalue decomposition, but with $D = \Sigma^2$. This shows that the eigenvalues and eigenvectors of $\hat{C}$ can be approximated with the singular values and right singular vector of $\hat{G}$.

The SVD decomposition can therefore be implemented as a replacement for the eigenvalue decomposition in this case due to the matrix being symmetric and positive definite[58]. This motivates the usage of Frequent Directions for Active Subspaces.

The rest of the procedure is as described in section 3.3, where a reduced version of $\hat{G}$ is stored. We therefore truncate it so that our $\hat{G}_{red}$ is given as

$$\hat{G}_{red} = \begin{bmatrix} \nabla_l g(x^{1,(l)}) & \nabla_l g(x^{2,(l)}) & \dots & \nabla_l g(x^{r,(l)}) \end{bmatrix}. \tag{5.9}$$

We then iterate through the rest of the samples in the training set, $i = r+1, \dots, n_{train}$. For every sample we update the reduced matrix so that

$$\hat{G}_{red} = \hat{V}_{red}\sqrt{\hat{\Sigma}_{red} - \hat{\lambda}_r}. \tag{5.10}$$

This gives us a $\hat{G}_{red}$ in descending order according to important directions, with the last column as the zero-vector. We then replace this with $\nabla_l g(x^{i,(l)})$, and continue with these two steps iteratively, eventually giving us our projection matrix

$$W_{proj} = \hat{V}_{red}. \tag{5.11}$$

Proper Orthogonal Decomposition We construct the Snapshot matrix from section 4 by setting the flattened pre-model outputs as columns, so that we have

$$\mathbf{S} = \begin{bmatrix} x^{1,(l)} & x^{2,(l)} & \dots & x^{N_{train},(l)}. \end{bmatrix}. \tag{5.12}$$

The snapshot matrix is then split up using singular value decomposition, giving us

$$\mathbf{S} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T. \tag{5.13}$$

By truncating $\mathbf{U}$, as done in [4], we gain the projection matrix

$$W_{proj} = U_r \in \mathbb{R}^{n_l \times r}, \tag{5.14}$$

consisting of the r first modes of the Snapshot matrix.

5.3.3 Input-output mapping

The input-output mapping consists of mapping the reduced output $z^j \in \mathbb{R}^r$ to the full model output $\hat{y}^j \in \mathbb{R}^C$, where C denotes the amount of classes, for any input j.

Polynomial Chaos Expansion We use the discrete approximation of the PCE model so that

$$\tilde{y} \approx \sum_{j=1}^{n_{train}} c_\alpha \psi_\alpha(z), \tag{5.15}$$

where $\psi_\alpha(z)$ are multivariate polynomial functions that are based on the probability density function $\rho(z)$. Due to our data sets being large, we assume that they follow a Gaussian distribution and therefore use Hermite polynomials as our ψ_α.

It is necessary to estimate the coefficients c_α. These can be found through solving the minimization problem

$$\min_{c_\alpha} \frac{1}{n_{train}} \sum_{j=1}^{n_{train}} ||\hat{y}^j - \sum_{|\alpha|=0}^{p} c_\alpha \psi_\alpha(z^j)||, \tag{5.16}$$

which can be done through linear regression or equivalently through solving

$$(\Psi^T \Psi)^{-1} \Psi^T \hat{X}. \tag{5.17}$$

For the sake of having a non-linear input-output mapping, the scheme with $p = 2$ is used.

Feedforward Neural Network Following our definition from section 2, we construct a FNN with one hidden layer.

The network input will be reduced output z, and so the amount of nodes for the input layer will be r. The amount of nodes in the hidden layer is chosen to be 20 from the work of [36]. The amount of nodes in the output layer will be $C = 10$, and the activation from the output layer will be linear.

The output from the net at any node i, $\tilde{y}_i$, will be given as

$$\tilde{y}_i = z_i^{(2)} = \sum_{k=0}^{C} w_{ik}^{(2)} z_k^{(1)} + b_i^{(2)} = \sum_{k=0}^{C} w_{ik}^{(2)} \sigma \left(\sum_{j=0}^{n_1} w_{kj}^{(1)} z_j^{(0)} + b_j^{(1)} \right) + b_i^{(2)}. \tag{5.18}$$

The activation function in the hidden layer could be any of the proposed functions from section 2.4.5. We have chosen the Softplus function, as done in [36]. As defined in section 2.4.5, it is given as

$$\sigma(x) = \frac{1}{\beta} \log(1 + e^{\beta x}). \tag{5.19}$$

5.3.4 Re-learning phase

The re-learning phase using knowledge distillation was described as in section 4, implemented with the same code and parameters as in [34, 8]. The teacher model for the knowledge distillation is the full model $\mathcal{ANN}$ for the respective data set, whilst the student models trained have been the different reduced neural network combinations, denoted in general as $\mathcal{ANN}_{red}$.

Denoting $\mathcal{L}_{KD}$ as the Kullback-Leibler loss function, $\mathcal{L}_{CE}$ as the cross-entropy loss function, $Q(x)$ to be the probability distribution of $\mathcal{ANN}$, and $P(x)$ to be the probability distribution of $\mathcal{ANN}_{red}$. The loss function for the knowledge distillation re-learning process is given as

$$\mathcal{L}(x^{(0)}, w) = \lambda \mathcal{L}_{KL} \left(Q(x^{(0)}), P(x^{(0)}), T \right) + (1 - \lambda) \mathcal{L}_{CE} \left(y, P(x^{(0)}) \right), \tag{5.20}$$

where y is a binary truth value, T is a temperature factor and λ is a regularization factor.

5.4 Numerical implementation

All source code used can be found in [32], if one wishes to follow the implementation.

The full model was taken from torchvision, initializing it as the fully trained model. As the fully trained model is adept at image recognition, the thought was that training the model for CIFAR-10 and SVHN would be quicker due to the weights perhaps being optimized towards some image recognition, and therefore the image recognition on a more specific data set (with fewer classes) would have similar weights. We re-trained it twice, creating two separate models to be used on each their data set: $\mathcal{ANN}_{\text{CIFAR}}$ and $\mathcal{ANN}_{\text{SVHN}}$.

All hyperparameters for training the full models were chosen according to Grad Student Descent[7]. The optimizer used was Stochastic Gradient Descent (SGD), with a learning rate of $\alpha = 0.0001$, momentum coefficient $\eta = 0.9$ and weight decay $\lambda = 5 \cdot 10^{-5}$. The batch size was $m_b = 64$ in both cases, and the total amount of epochs were set to be 100. The loss function used was Cross-Entropy Loss.

The construction of the pre- and post-model, $\mathcal{ANN}_{\text{pre}}$ and $\mathcal{ANN}_{\text{post}}$, was done based on the source code of the full model VGG-16 [42], with modifications so that no flattening was done in the last layer of the pre-model. In order to ensure that the composition of the pre- and post-model still gave the correct output, we ran tests that computed $\hat{y} = \mathcal{ANN}_{\text{full}(x)}$ and $\hat{y} = \mathcal{ANN}_{\text{post}}(\mathcal{ANN}_{\text{pre}(x)})$ simultaneously for the testing set, and compared the accuracy. Generally they performed equally good, with perhaps a difference in accuracy to the order 10^{-3}, which we gathered to be some round-off error in an activation function and therefore negligent.

Construction of POD was done using the PyTorch library.

Construction of Active Subspaces was done using a combination PyTorch and the ATHENA Active Subspaces library [44]. The spatial gradients required for the function are computed using automatic differentiation, which is done through PyTorch's autograd function.

Construction of the Feedforward Neural Network was done in PyTorch.

Construction of the Polynomial Chaos Expansion was done using PyTorch tensors, and SciPy[48] for the orthogonal basis functions. Additionally, linear regression from the sklearn[47] library was implemented to find the coefficients of the multivariate polynomials in PCE.

The re-training using knowledge distillation was implemented from the code of [34, 8]. In order to pass the reduced neural network through the knowledge distillations, the different steps of the reduction process were set in as layers of a sequential model in PyTorch.

The Adam optimizer was used for the re-training process, with a step size $\alpha = 10^{-4}$ for the pre-model layers, and $\alpha = 10^{-5}$ for the reduction layer and the input-output mapping. The temperature factor was set to be $T = 1$, and the regularization factor was set to be $\lambda = 0.1$. The re-training was carried out for 10 epochs. This took approximately 1 hour per model, when using 32 CPUs.

It is worth stating that a lot of the numerical procedures from this thesis are time consuming and computationally expensive. High Performance computing has been utilized for the majority of the calculations undertaken, leading to a vastly smaller computation time than for regular devices.

6 Results

The results will initially be presented as comparisons between the full model trained with the augmented data, and the full model trained with the original data. Some preliminary values gained when conducting model reduction will also be investigated. A quantitative analysis of various parameters in the model reduction will follow, and their impact on the accuracy and function of the reduced model. Additionally, there will be a qualitative analysis of various combinations and the results gained from their predictions.

All the different combinations of reduction layers and input-output mappings have been implemented successfully, and will be examined in detail. Both the results before and after knowledge distillation are of interest. No direct comparison between the performance of the reduced net between SVHN and CIFAR will be made, as they solved different classification tasks. Any critical difference in performance can be blamed on the choice of the neural network or hyperparameters, as opposed to the reduction methods.

6.1 Full model results

The full model is the fundamental model from which we are doing the model reduction. Due to the weights being inherited for the pre-model and the input-output layer mapping to the full model output, it is obvious that the positive and negative qualities from the full model will be inherited by the reduced model. It is therefore crucial to take a deep-dive into the specifics of the full model.

VGG-16 was trained with the same hyperparameters for CIFAR, SVHN, CIFAR-augm and SVHN-augm. For each epoch, the training loss and validation loss was calculated. Plotting the training and validation loss can give important information about the model behavior. In order to compare the model trained on augmented, normalized data with the model trained on the original data, the two are be plotted against each other.

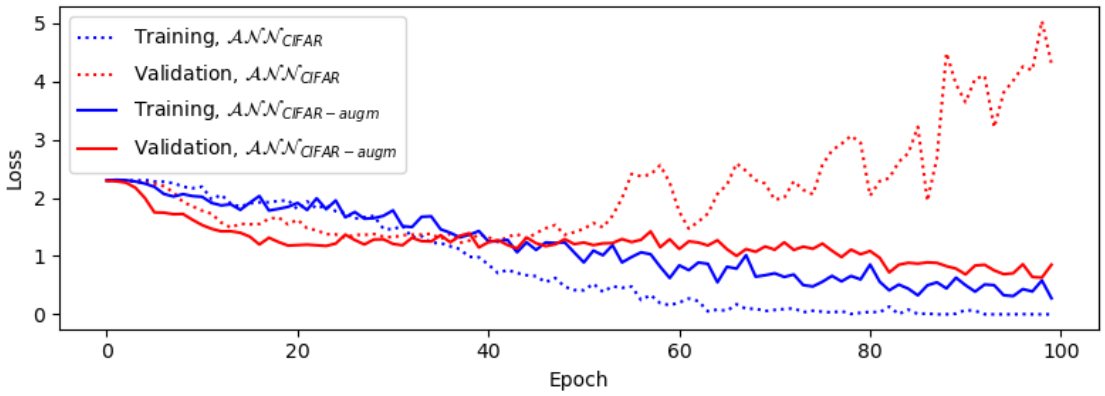


Figure 6.1: Training loss and validation loss of the CIFAR dataset.

Both figure 6.1 and figure 6.2 show that there are clear tendencies to over-fitting in the models that are trained on the original data. In both cases the validation loss is significantly worse than the training loss. The models trained on augmented and normalized data, on the other hand, have validation and training loss curves that are improving synchronized. These models are therefore more viable as they, despite having a lower training accuracy, have a greater validation accuracy. The validation-loss curves therefore suggest that models trained on normalized, augmented data may give more accurate predictions.

Table 2 shows some quantitative results for the full models. The accuracy is the accuracy when sending the testing set through the network. The size is the amount of bytes it took to save the model as a pth file, and the time is the amount of time it took to send the testing set through the network. The testing set of SVHN is significantly larger than the CIFAR testing set, which

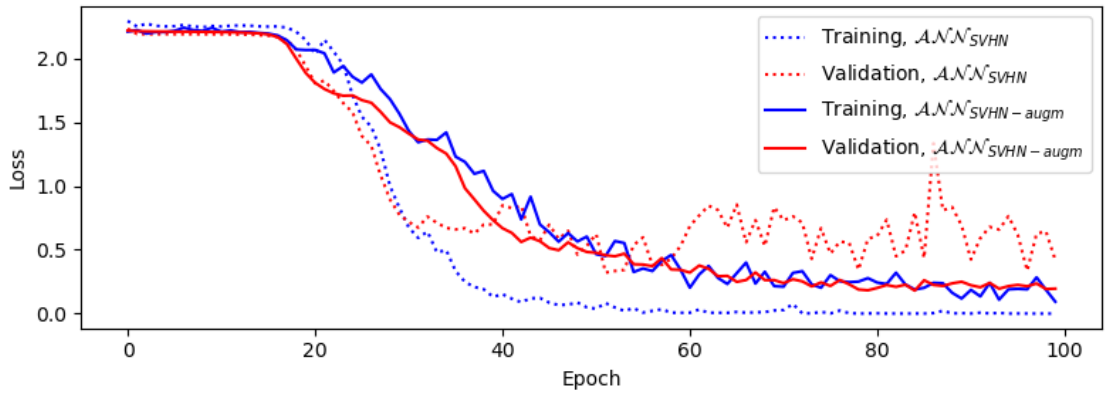


Figure 6.2: Training loss and validation loss of the SVHN dataset.

$\mathcal{ANN}_{\text{model}}$	Accuracy	Storage size	Time (s)
CIFAR	66.99%	1 GB	22
CIFAR-augm	79.77%	1 GB	22
SVHN	89.87%	1 GB	50
SVHN-augm	91.85%	1 GB	50

Table 2: Quantitative results from the full model.

explains why the computation time was much slower for this instance. All models have the same amount of parameters, and therefore take up the same space on a computer.

The accuracy of the CIFAR-trained models are, in both instances, significantly worse than the SVHN-trained models. However, CIFAR consists of different vehicles and animals and can be deemed to be more complex than SVHN, consisting only of digits. It is therefore not surprising that it is more difficult to classify CIFAR, and especially in the cases where the data is not normalized nor augmented.

Already from the validation-loss curves and the accuracy, we are seeing tendencies that the models trained on the original data are performing below desired capacity.

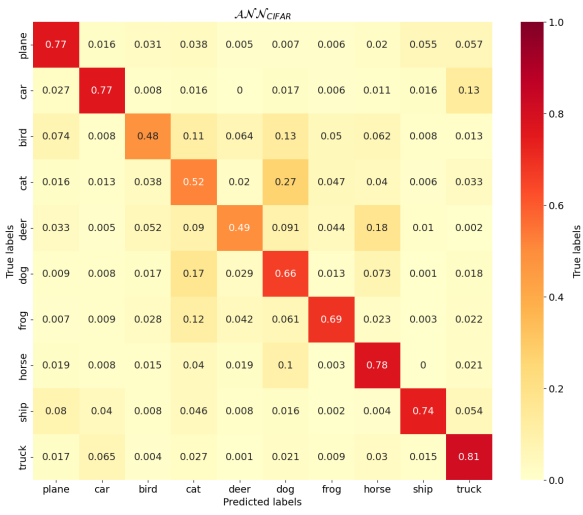


Figure 6.3: Confusion matrix from the testing of $\mathcal{ANN}_{\text{CIFAR}}$.

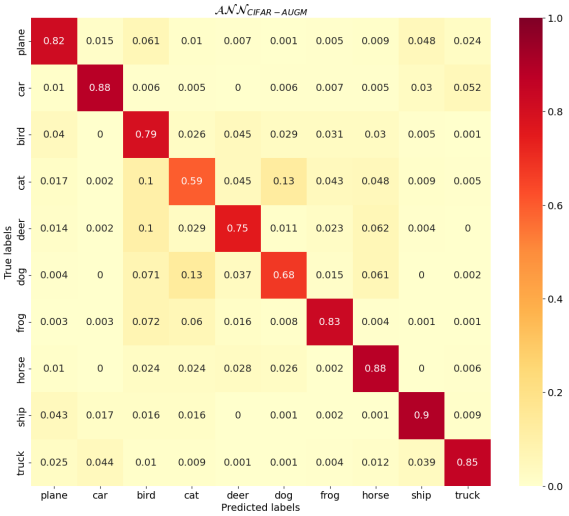


Figure 6.4: Confusion matrix from the testing of $\mathcal{ANN}_{\text{CIFAR-augm}}$.

Figure 6.4, 6.3, 6.6 and 6.5 visualize the confusion matrix for the respective data sets during evaluation. The confusion matrix pairs the predicted labels with the true labels, to give a rep-

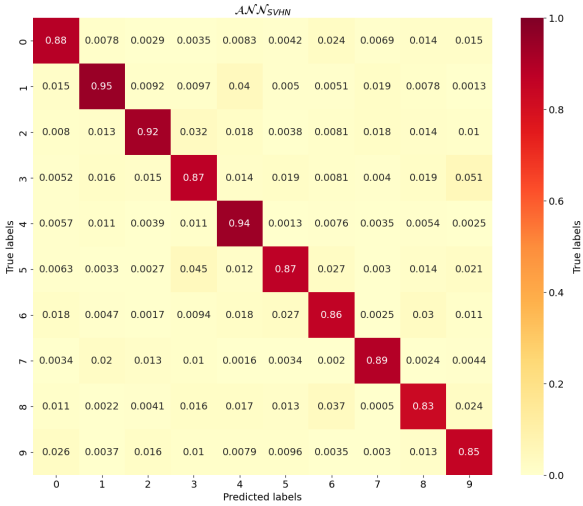


Figure 6.5: Confusion matrix from the testing of $\mathcal{ANN}_{\text{SVHN}}$.

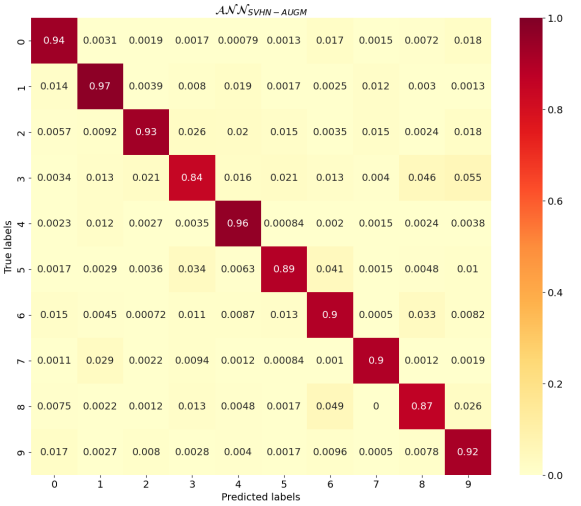


Figure 6.6: Confusion matrix from the testing of $\mathcal{ANN}_{\text{SVHN-augm}}$.

resentation of how accurate the predictions are for each respective class. Here we see that all the models are generally able to predict the different classes, with the models for CIFAR performing noticeably worse than the models for SVHN. Furthermore, we see that the accuracy is much worse for the models without data augmentation and normalization. The general pattern of the confusion matrices showed here is a strong diagonal tendency, which is desirable as it means that most predictions are correct.

It is by far, as already found in table 2, the model trained on the original CIFAR-data that performs the poorest. When looking at figure 6.4 we can see that some of the largest mistakes it makes are between similar classes in the data set. The model is, for example, not good at making a distinction between a cat and a dog. It is also struggling at distinguishing horses from deers and trucks from cars, and generally has a low accuracy score for cats, birds and deers. It seems to almost randomly guess at other categories when presented with a bird, which is problematic, though it shows a strong tendency to at least recognize that birds must be in the animal category. On the other hand, it makes obviously wrong mistakes such as mistaking a deer for a car or a ship for a horse.

Figure 6.3 shows large improvements in the classification done by the model. Although the largest misclassifications, namely distinguishing a cat from a dog, are still prevalent, the model is able to classify the samples of all classes with an accuracy larger than 55%, and the desirable diagonal tendencies of the confusion matrix are much more prevalent here. Furthermore, it seems to insinuate that the classification of a cat is perhaps one of the harder tasks of the data set due to similarities with other classes, which there may be some truth in.

Figure 6.6 and 6.5 are very similar to each other, with a strong diagonal element and no clear outliers in terms of misclassifications. From table 2 we saw that they both had a high accuracy, with the model trained on augmented data being slightly more accurate. It is interesting to see that there are some minuscule differences in the diagonal elements of the confusion matrices. For example, $\mathcal{ANN}_{SVHN-augm}$ struggles more with classifying the cipher 3. However the rest of the diagonal elements, especially in the lower part of the matrix, are generally better classified leading to an overall better classification score.

Due to the large discrepancy between the predictions of the model trained on the original data, and the correct labels of the testing set, it is viable to discard the models trained on the original data. A poorly performing model will lead to poorly performing reduced models, which is not desirable. For the rest of the numerical procedures in this thesis we will therefore only use $\mathcal{ANN}_{CIFAR-augm}$ and $\mathcal{ANN}_{SVHN-augm}$. Any usage of the data set names CIFAR and SVHN will refer to the augmented and normalized data, unless specified.

6.2 Incremental results

The incremental results contain results from the pre-model layer and the reduction layer. Results from the input-output mapping are omitted from the incremental results as they provide classifications, and are therefore comparable with the results after re-learning.

6.2.1 Pre-model results

From section 5, the choice of cutoff layer was discussed. As mentioned, the works of [36, 13] choose a cutoff layer that is not in direct contact with the max-pooling layers. This is done with the intention of using a cutoff that reduces the amount of parameters in the model.

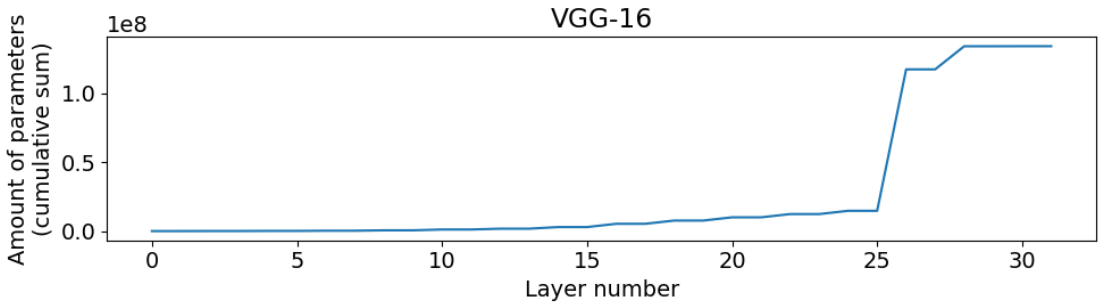


Figure 6.7: The amount of parameters per layer of the sequential version of the VGG-16 model.

Figure 6.7 depicts the cumulative amount of parameters in our modified version of VGG-16. Especially in the latter layers of the model, one can see that the amount of parameters is extremely high. A high amount of parameters takes up a lot of space storage-wise, and is time-consuming to train. Choosing a cutoff layer before the accumulation of parameters becomes too high, i.e. around layer 15 or somewhere in the surrounding area, would therefore be ideal. However, there is a trade-off when splitting the full model into a pre- and post-model too early. The various layers that are dependent on the numerous parameters are feature extractors, and can gather important non-linear patterns from the data set. By choosing a split very early on the model, one may risk losing the non-linear connections gathered from the different filters.

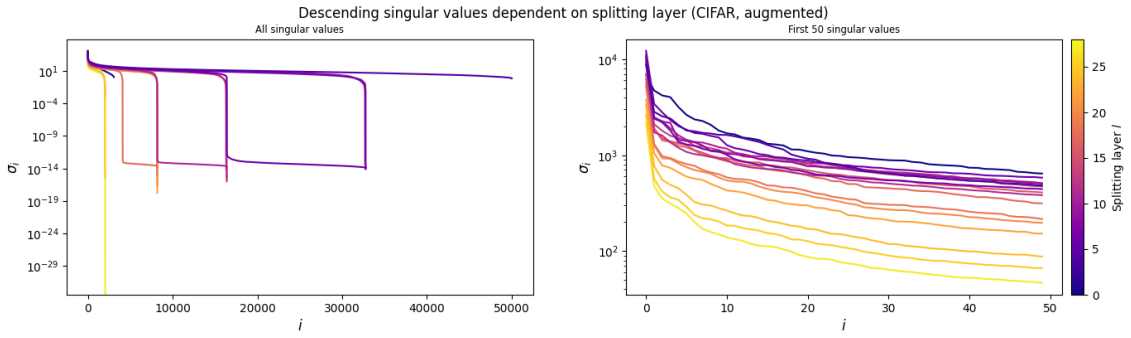


Figure 6.8: SVD of the output of VGG-16 at various layers, for the CIFAR training set.

Figure 6.8 and 6.9 show the descending singular values according to the cut-off layer chosen for the pre-model. The color bar on the right denotes the cutoff layer. The plots on the left show the entire span of the singular values. Here one can also see that there are fewer singular values for the latter layers, meaning that the outputs are of smaller dimensions. The plots on the right show the first 50 singular values. This is to see the impact of the cut-off layer on the singular values on a more close-up scale. This is also the maximum reduction dimension tested in this thesis, and so the directions belonging to the rest of the rest of the singular values will anyways be discarded during model reduction.

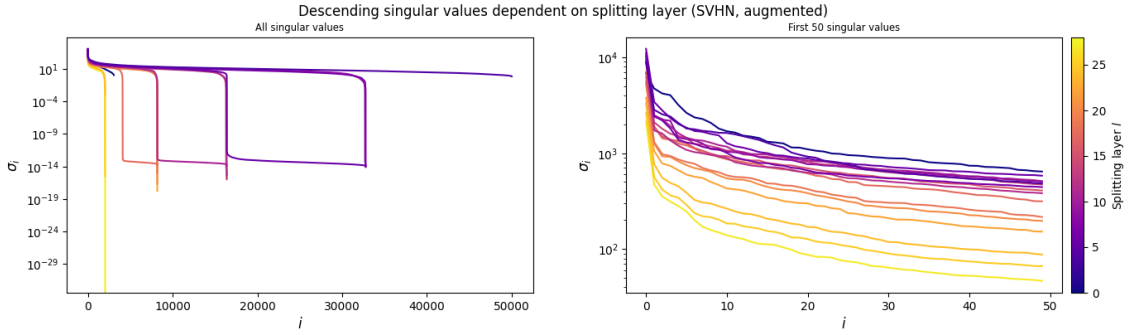


Figure 6.9: SVD of the output of VGG-16 at various layers, for the SVHN training set.

In both figure 6.8 and 6.9, the singular values of the later layers in the model are the ones that descend the quickest. A quick descent is important for model order reduction. In both Active Subspaces and Proper Orthogonal Decomposition, we reduce the model by projecting it onto the most important directions of the high-dimensional data. A quick descent of singular values signifies that some directions are indeed much more important than others. A quick descent can imply that a model reduction is appropriate, as not too much information will be lost in the process. The SVD figures thereby show that a cutoff for a latter layer may give better results due to the singular values descending faster.

6.2.2 Projection matrix results

We want to illustrate how much information is lost when projecting the input to a lower dimension. This is mainly to see whether there is a huge information loss in either Active Subspaces or Proper Orthogonal Decomposition. There is no point in utilizing a method that discards a lot of information during the model reduction.

Our reduced pre-model output is given as $z = W_{proj}^T x^{(l)}$, whilst our pre-model output is given as $x^{(l)}$. We aim to get some qualitative view of the difference between $x^{(l)}$ and the reduced variable z.

Recall that our projection matrices will be orthogonal, meaning $W_{proj} W_{proj}^T = I$. This means that $\tilde{x}^{(l)} = W_{proj} z$ should be equivalent to $x^{(l)}$, however due to information loss there will be a difference.

Looking at the difference relative the initial value of $x^{(l)}$. The relative difference is given as

$$\frac{||x^{(l)} - \tilde{x}^{(l)}||}{||x^{(l)}||} = \frac{||x^{(l)} - W_{proj} W_{proj}^T x^{(l)}||}{||x^{(l)}||} = \frac{||(I - W_{proj} W_{proj}^T) x^{(l)}||}{||x^{(l)}||}. \tag{6.1}$$

Using the Cauchy-Schwartz inequality here, we get the upper bound

$$\frac{||x^{(l)} - \tilde{x}^{(l)}||}{||x^{(l)}||} \leq \frac{||I - W_{proj} W_{proj}^T|| \cdot ||x^{(l)}||}{||x^{(l)}||} = ||I - W_{proj} W_{proj}^T||. \tag{6.2}$$

We therefore wish to use $||I - W_{proj} W_{proj}^T||$ as a relative measure of the upper bound of the information loss, where the norm $||.||$ can be chosen according to preference.

The upper bound of the projection matrix is plotted as a function of the cutoff value, the reduction dimension value r, and the sample amount in figure 6.11 and 6.10. The norm used is the Frobenius norm. For the instances with the upper bound as a function of the cutoff and the r value, the labels are omitted due to the curves being identical. In the figures where the upper bound varies according to n_{train}, we see that two cases of POD and AS produces a slightly different upper bound of information loss. However, one can see that the scale is in the area of 10^{-10}, meaning the difference is almost insignificant.

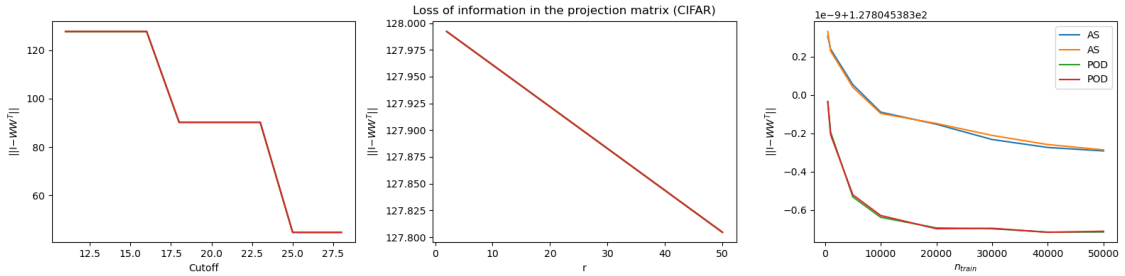


Figure 6.10: The loss of information in the projection matrix according to hyperparameters, for CIFAR.

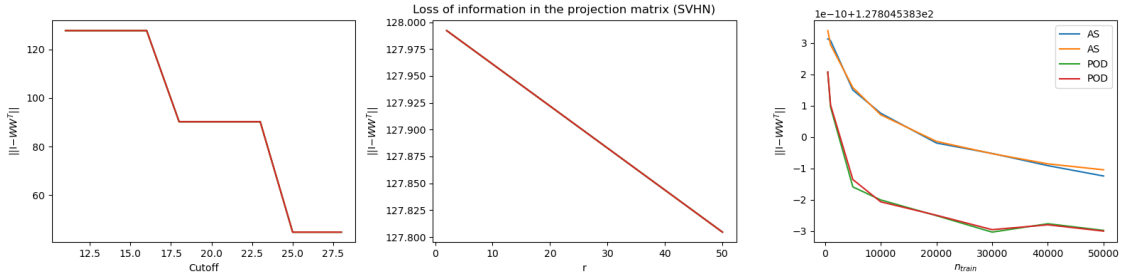


Figure 6.11: The loss of information in the projection matrix according to hyperparameters, for SVHN.

It is worth noting that the zig-zag patterns in the bound as a function of the cutoff are taking place at the max-pool layers. This makes sense, as the max-pool layers halve the output. The norm is not a relative norm, so that a matrix of smaller dimensions will give a smaller norm. The loss of information as a function of r is linear, however the change in the y-axis is not very steep. This suggests that r does not really have a big impact on the information loss of the projection matrix, as one always retains the most important direction regardless.

The equivalence of information loss in the projection matrices stemming from both AS and POD suggests that the two methods are projecting the model onto the same subspace. Both methods are utilizing singular value decomposition, but it is done on different matrices: POD is completely data-driven and relying on the pre-model output, whilst AS is relying on the gradient of the output of the full model.

6.3 Quantitative results

Note that this is a study of proof of concept. There has therefore not been a large focus on optimizing the numerical methods utilized in the model reduction and training part. The accuracy, computational space and time spent may vary dependent on hyperparameters chosen, packages implemented and so on, so that the results presented here may vary considerably from the results of those with a completely approach to coding. The results should therefore be taken with a grain of salt: a specific method can perform worse in this exact setting with these exact parameters, but this doesn't necessarily mean that it will always perform worse.

The figures of this subsection all show quantitative results of the model reduction according to cutoff layer l, reduction dimension r, and the amount of training samples $n_{samples}$ used for constructing the projection matrix and input-output layer in the neural network. The metrics of importance that were presumed to be affected by these parameters were deemed to be accuracy, the running time and the storage space for the model.

The baseline parameters for the reduced models are chosen to be

- $n_{train} = 50000$,

- $r = 50$,
- $l = 16$.

All the figures contain the four different combinations of reduced model types (AS + PCE, AS + FNN, POD + PCE, POD + FNN) with the baseline parameters kept constant, except for the variable plotted on the x-axis. The aim is to see trends and coherences between the parameters of the reduced model and its respective output.

6.3.1 Before re-learning

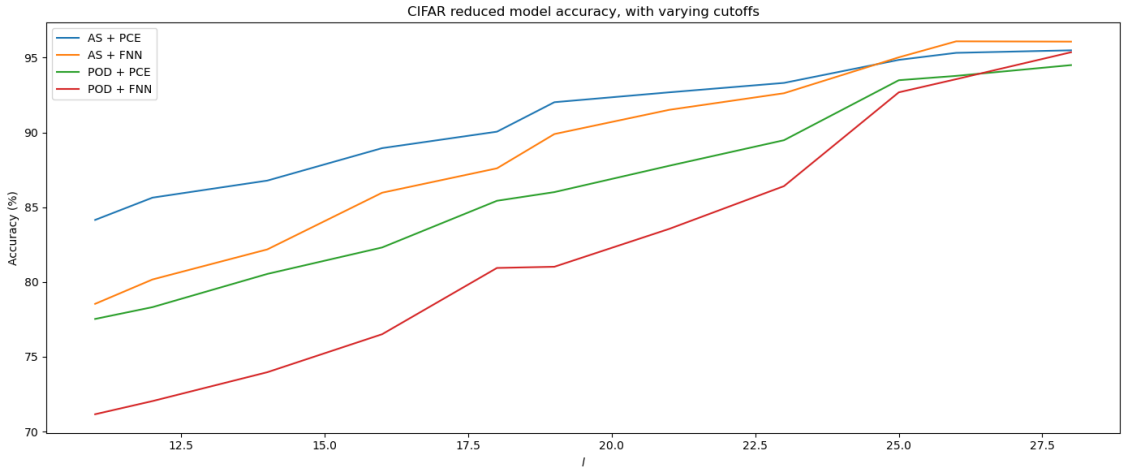


Figure 6.12: Reduced model accuracy according to the cutoff, for the CIFAR data set.

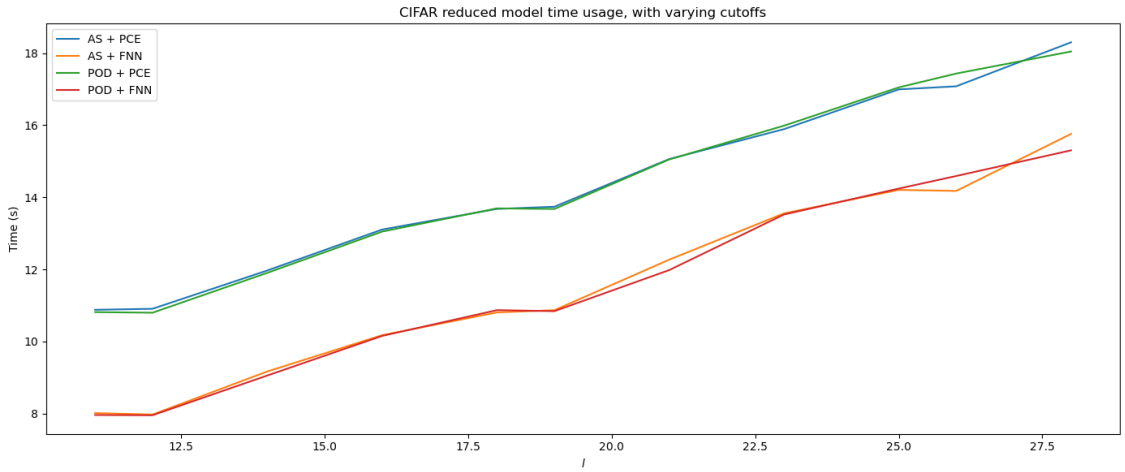


Figure 6.13: Reduced model time usage according to the cutoff, for the CIFAR data set.

In all figures concerning accuracy i.e. figure 6.12, 6.12, 6.18, 6.18, 6.27 and 6.24, we can see that there is a clear order of accuracy between the different reduced models. The models that use AS to find the projection matrix are generally performing better than the models using POD. Furthermore, the models using PCE as an input-output map have a higher accuracy than those using FNN. However, in all figures the "fuller" reduced models i.e. the reduced models with a larger r value or taken at a later cut-off have a more similar level of accuracy than the "less full" models.

From figure 6.15 and 6.12 we observe that the relationship between the accuracy of the models and the cutoff almost look linear, i.e. a later cutoff in the model will give a proportionally better

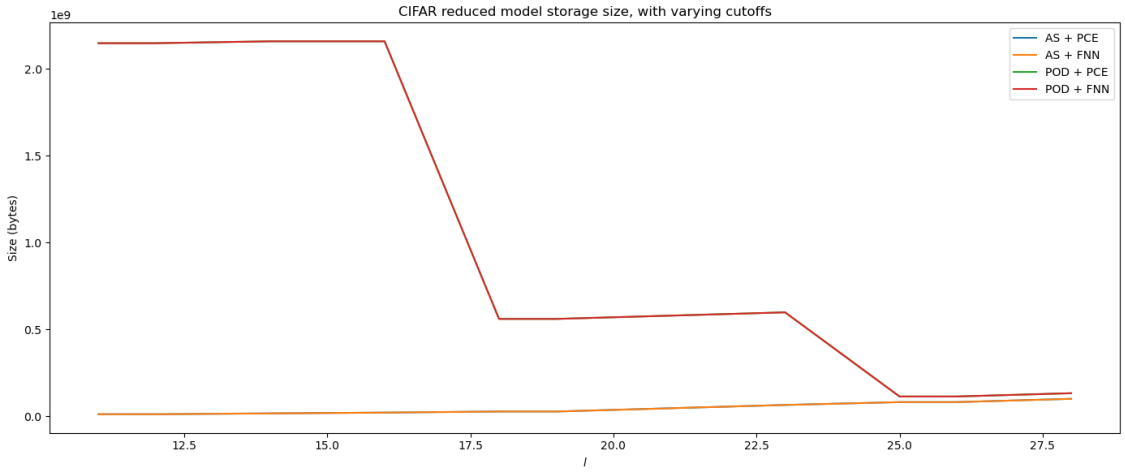


Figure 6.14: Reduced model storage size according to the cutoff, for the CIFAR data set.

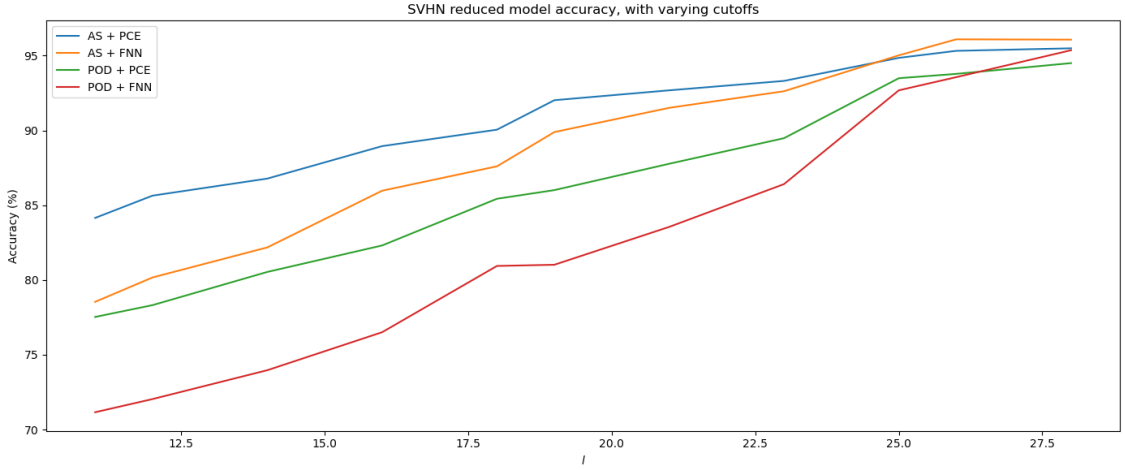


Figure 6.15: Reduced model accuracy according to the cutoff, for the SVHN data set.

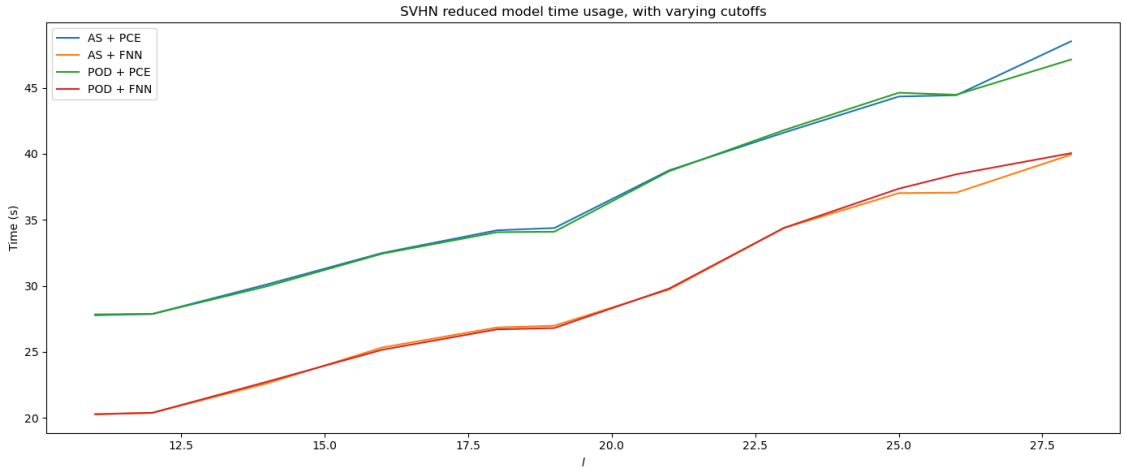


Figure 6.16: Reduced model time usage according to the cutoff, for the SVHN data set.

accuracy score. The relationships between the accuracy and the reduction dimension r as shown in figure 6.21 and 6.18, on the other hand, appear to be stagnating. It seems as if all accuracy, when r is increased, converges to some high percentage. This suggests that there is no point in

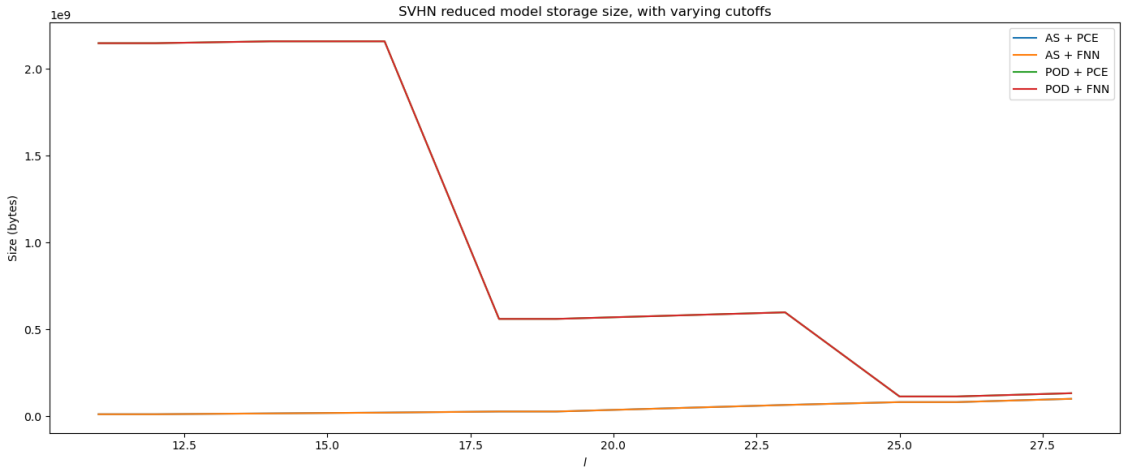


Figure 6.17: Reduced model storage size according to the cutoff, for the SVHN data set.

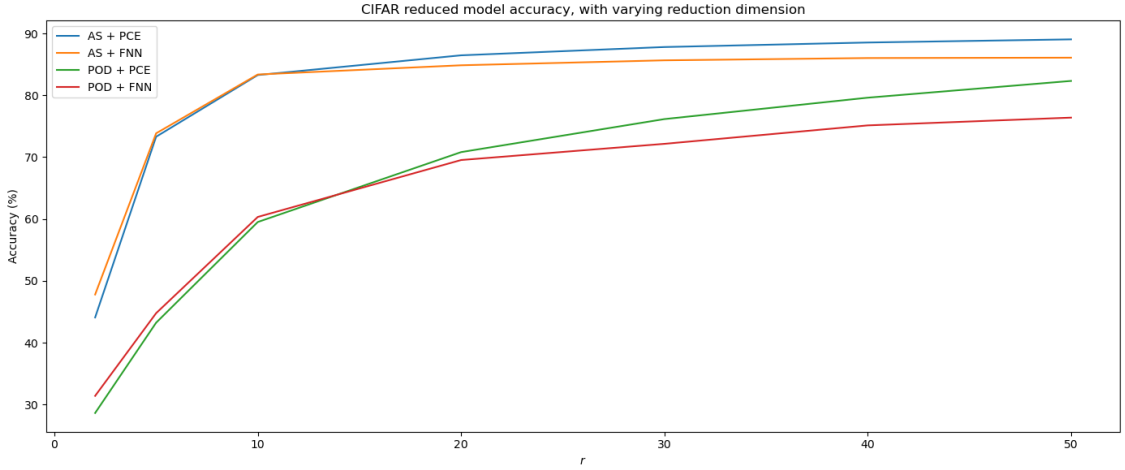


Figure 6.18: Reduced model accuracy according to the reduction dimension, for the CIFAR data set.

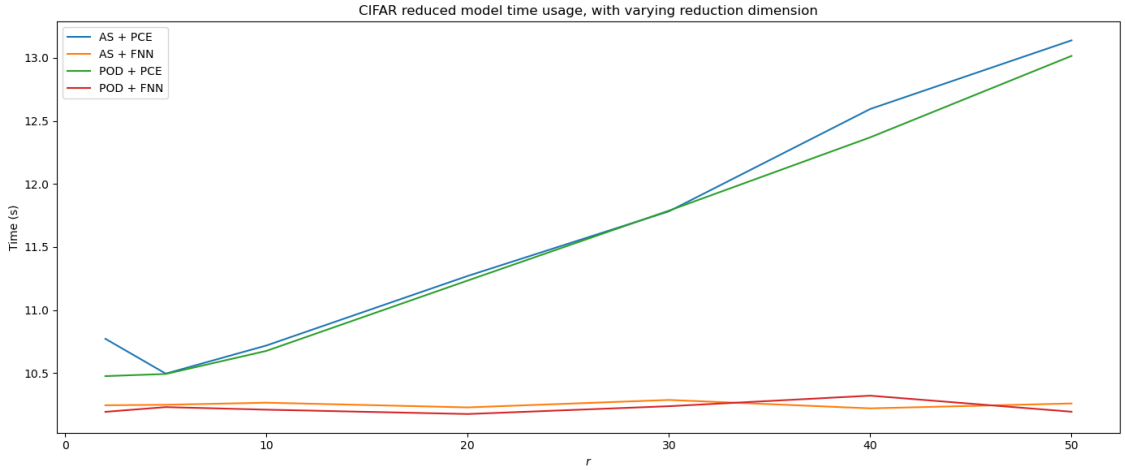


Figure 6.19: Reduced model time usage according to the reduction dimension, for the CIFAR data set.

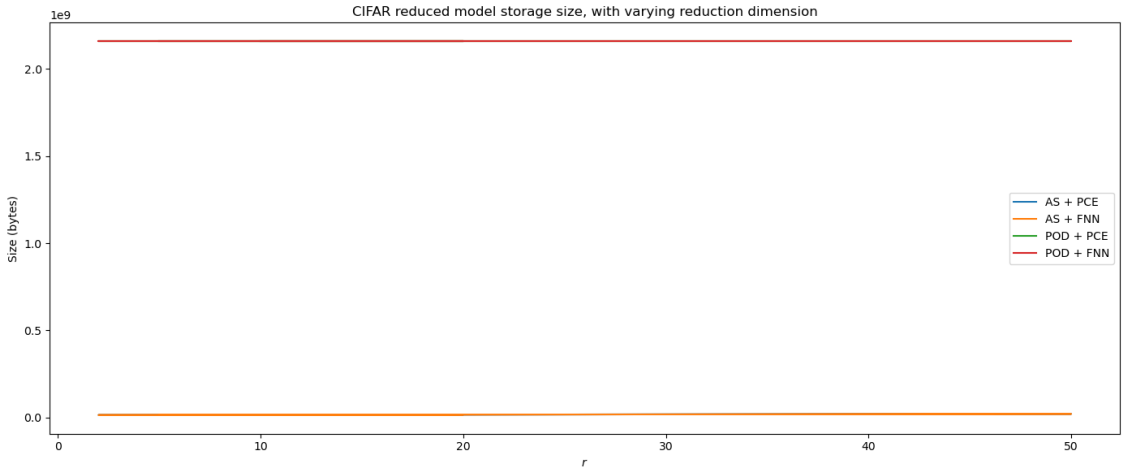


Figure 6.20: Reduced model storage size according to the reduction dimension, for the CIFAR data set.

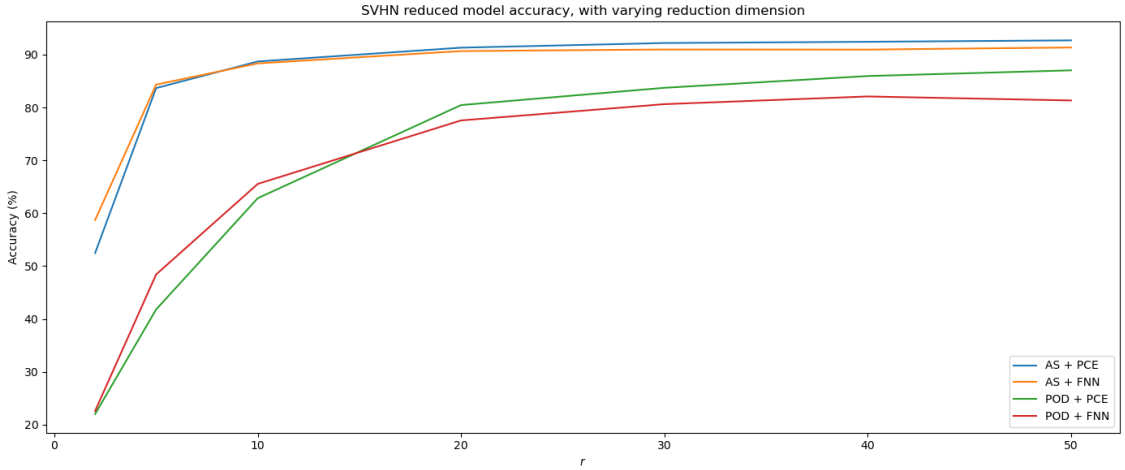


Figure 6.21: Reduced model accuracy according to the reduction dimension, for the SVHN dataset.

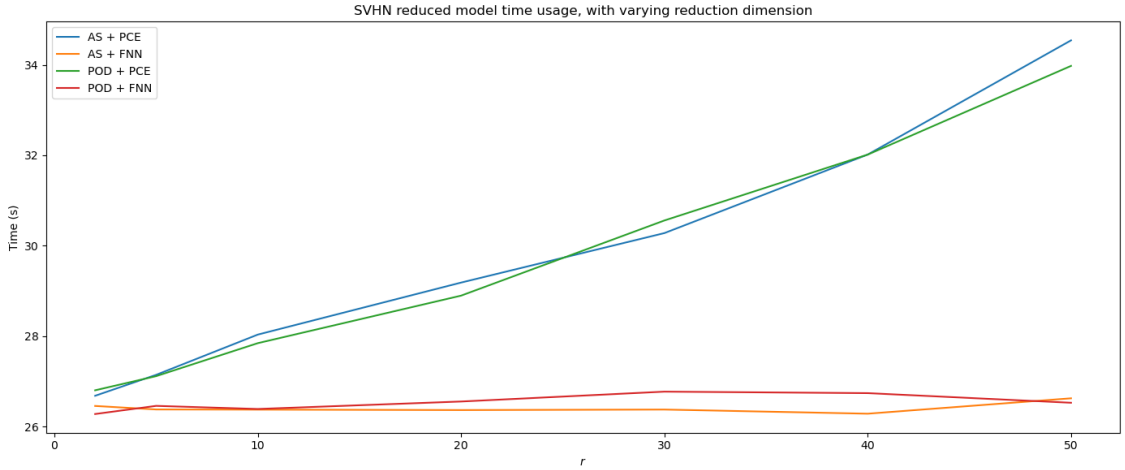


Figure 6.22: Reduced model time usage according to the reduction dimension, for the SVHN data set.

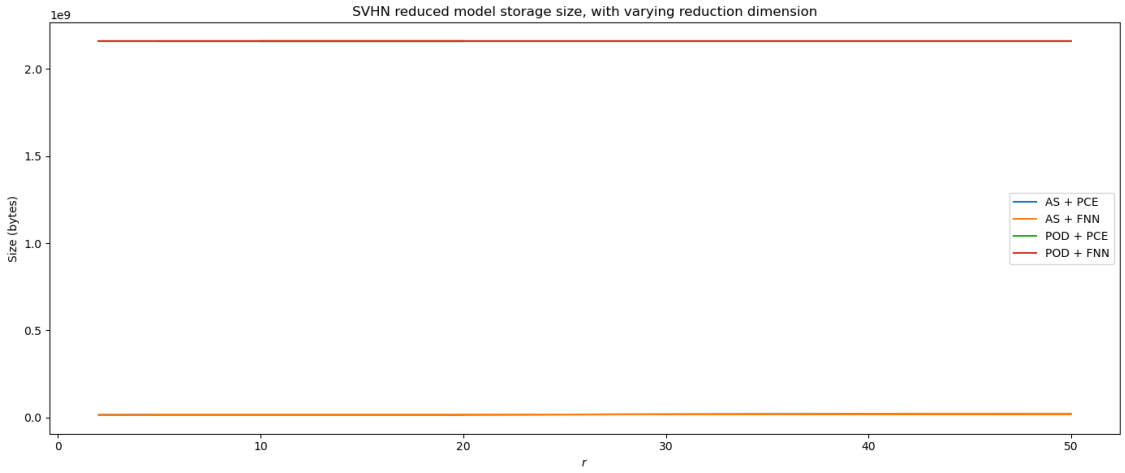


Figure 6.23: Reduced model storage size according to the reduction dimension, for the SVHN data set.

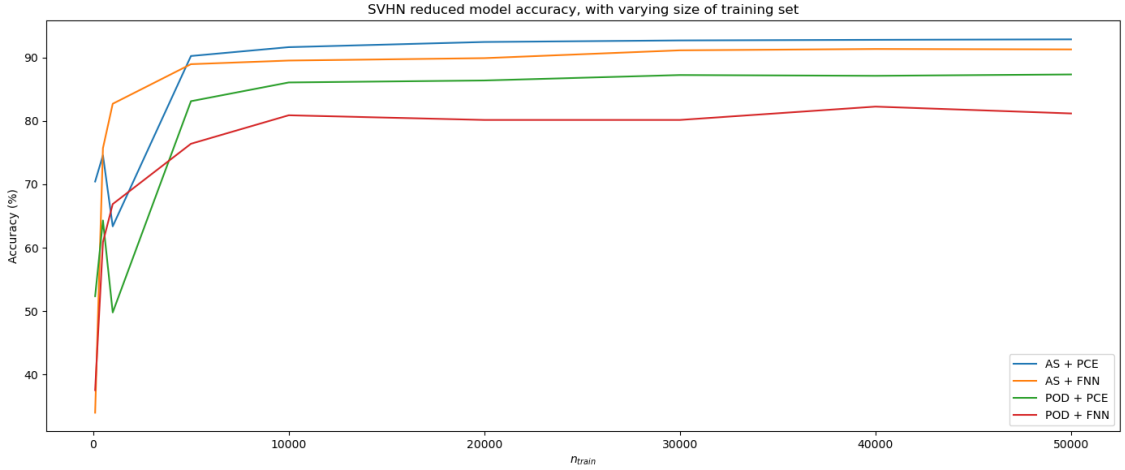


Figure 6.24: Reduced model accuracy according to the size of the training set, for the SVHN data set.

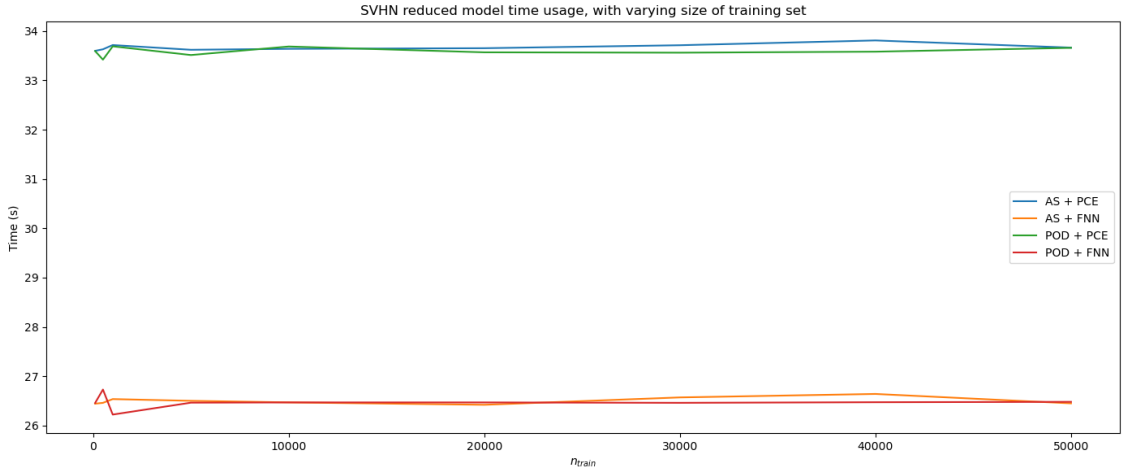


Figure 6.25: Reduced model time usage according to the size of the training set, for the SVHN dataset.

Figure 6.26: Reduced model storage size according to the size of the training set, for the SVHN data set.

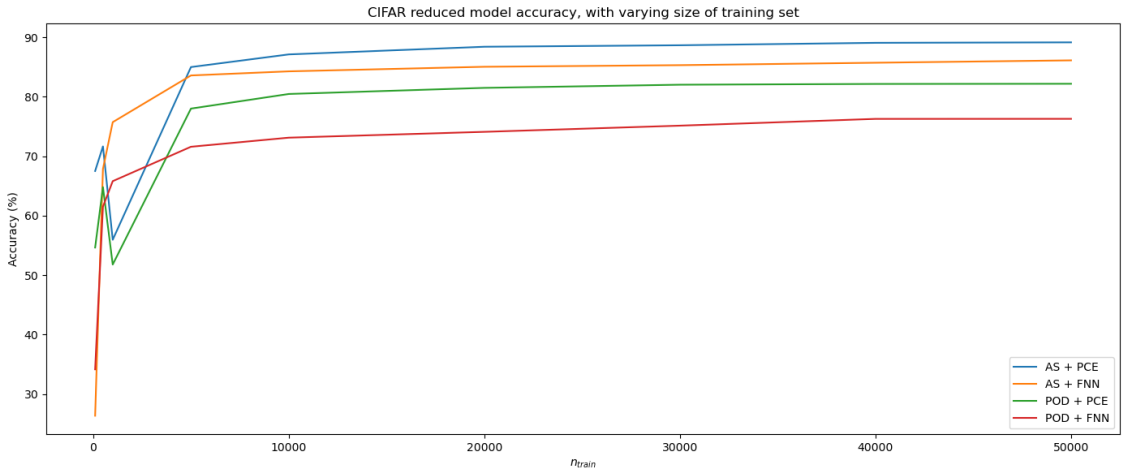


Figure 6.27: Reduced model accuracy according to the size of the training set, for the CIFAR data set.

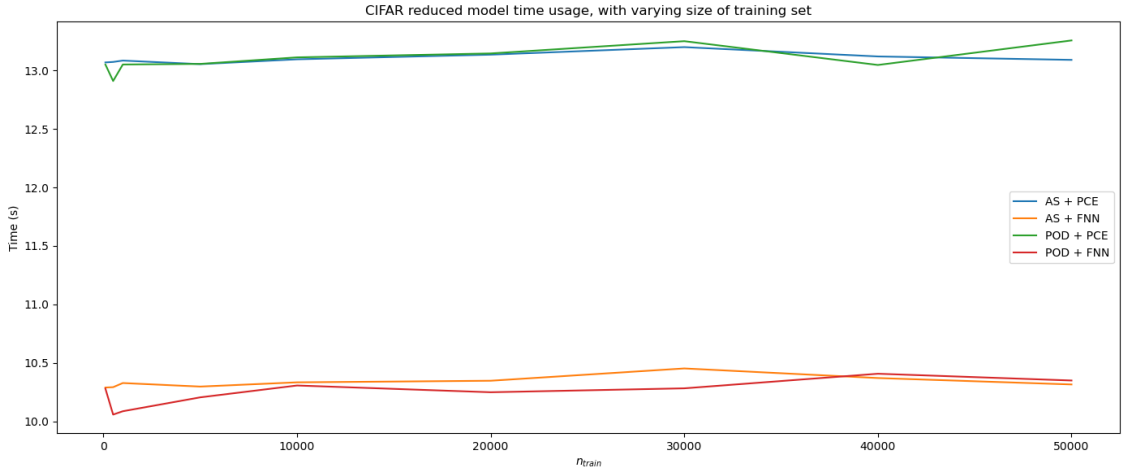


Figure 6.28: Reduced model time usage according to the size of the training set, for the CIFAR data set.

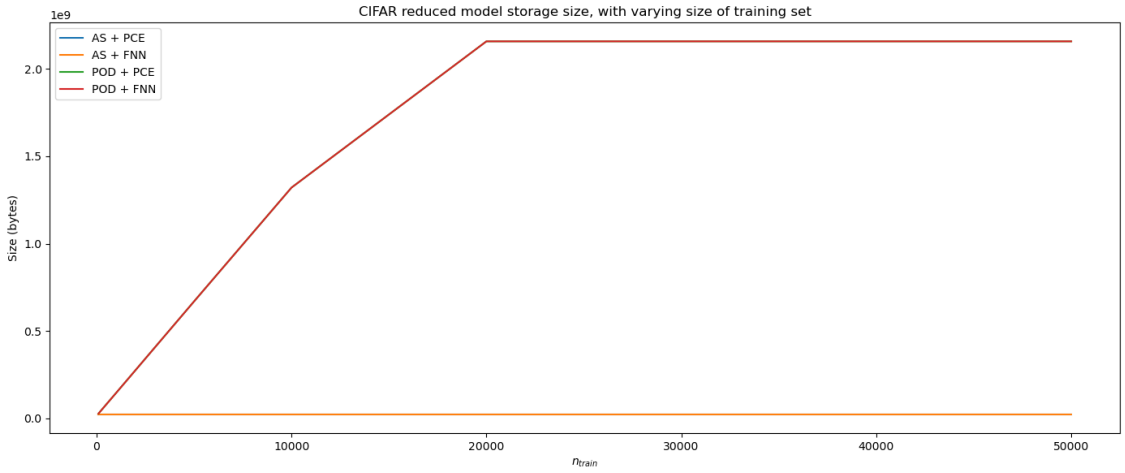


Figure 6.29: Reduced model storage size according to the size of the training set, for the CIFAR data set.

increasing the r value unless it is extraordinarily small and severely damaging the accuracy of the neural network altogether.

Figures 6.24 and 6.27 suggest that the amount of training samples used for constructing the reduced neural network does not particularly influence its accuracy. This is of course restricted to a certain amount: The accuracy in both plots are extremely low when using as few as 100 samples. We see in both plots that there are sharp increases and decreases in the area between 100 samples and 5000 samples, but one can attribute this to coincidence according to the samples used. In some cases one may have a selection of well-balanced samples that represent a large part of the entire data set, whilst in other cases one may have a huge class imbalance in the training set, leading to a poorer accuracy. Generally, it seems that having 5000 samples in the training set or having 50000 does not impact the accuracy at all, when constructing the reduced model.

Figures 6.13 and 6.16 show an almost linear relationship with the cutoff value and the time taken to send the training set through the model. The relationship shows that the reduced model is quicker at computing predictions than the full model, and motivates reduction at an earlier cutoff.

Figures 6.14, 6.17, 6.20, 6.23, 6.29 and 6.26 imply that the biggest impact on storage size is the method for finding the projection matrix. In all cases, the projection matrix found through POD seems to take up the most space. Figures 6.29 and 6.26 show that the storage space used is influenced in POD by the amount of samples in the training set, up until a fixed amount of samples. Figures 6.14 and 6.17 show that the storage space used by the reduced model decreases according to the cutoff layer chosen. This makes sense, as the dimensions in the earlier layers of VGG-16 are extremely high, and so the flattened output will be large, giving us huge projection matrices and a lot of parameters when mapping the reduced variable onto the solution space. The dimensions in the latter layers are much lower, giving less parameters to save in the projection matrix and the input-output mapping. Also, figures 6.14 and 6.17 show that the storage space decreases at max-pooling layers, thus indicating that choosing an appropriate pre-model cutoff may be one of the parameters affecting the storage space the most.

Figures 6.19 and 6.22 suggest a linear relationship between the time usage of PCE and the reduction dimension r. Recall that the amount of polynomials in PCE is dependent on the order of the polynomials and r, meaning this does indeed have a strong relationship. A larger r leads to more polynomials in PCE, thus yielding a greater computation time for the reduced model. The reduction dimension, however, does not affect the amount of layers in the FNN nor the size of the hidden layer (with the exception of the input), and so the computation time of FNNs do not seem to be influenced by r at all.

6.3.2 After re-learning

The time and storage aspect will not be taken into consideration when comparing the post-re-learning results with the pre-re-learning results. The time taken to run the test set through the models, as well as the storage space taken up by the models trained by knowledge distillation, are equivalent to those presented in section 5. Furthermore, different CPUs have been used for the different tests and so any difference in time spent can be the result of the change in CPUs.

All different reduced models have been re-trained with the same parameters and amount of epochs, using knowledge distillations. The new accuracy as a function of cutoff, reduction dimension and samples in the training set will be presented, as well as the percentage-wise change in accuracy.

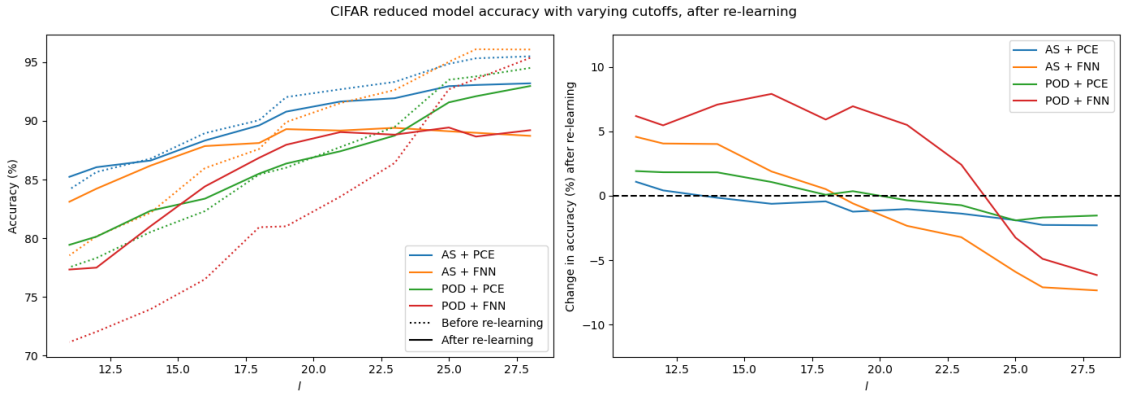


Figure 6.30: CIFAR reduced model accuracy according to cutoff, after re-learning.

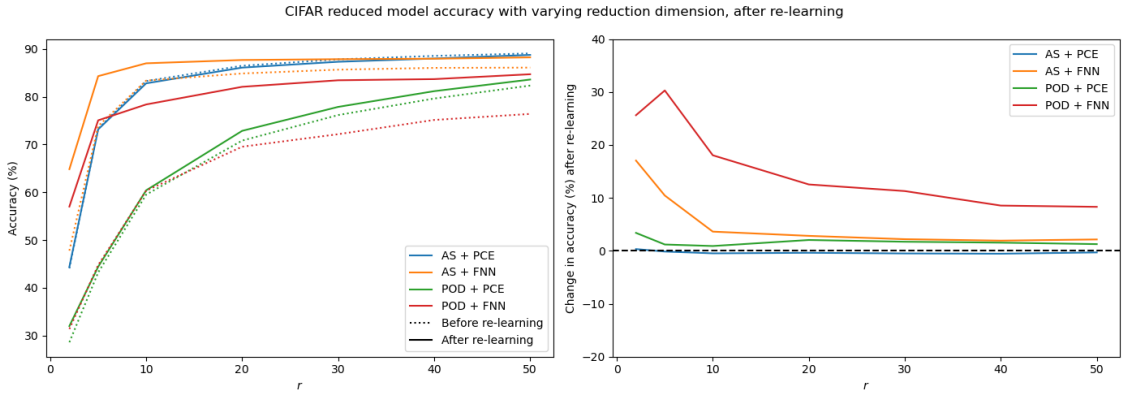


Figure 6.31: CIFAR reduced model accuracy according to reduction dimension, after re-learning.

From figures 6.30, 6.33, 6.31, 6.34, 6.32 and 6.35 we can see a clear trend in the reduced model type that improve the most through the re-training. In all cases it can be observed that the accuracy does not change much for the reduced models that are dependent on PCE. It should be noted that they are already of very high accuracy, so that when the models dependent on FNN are improved through re-learning it leads to all the reduced models being of around the same accuracy. Ultimately, the figures suggest that re-learning is only useful in cases of subpar accuracy.

Figures 6.30 and 6.33 show the increase or decrease in accuracy according to cutoff, as a result of knowledge distillation. Immediately one can observe that knowledge distillation will increase the accuracy for models with an early cutoff layer, and decrease the accuracy for models with a late cutoff layer. The plots to the left show that for the models with the late cutoff layer, the accuracy is already quite high. The decrease in accuracy can therefore be due to over-fitting. The plots suggest that there is no need for re-learning when choosing cut-off layers that are after layer 18 or so, as they will already be fairly accurate.

In figures 6.31 and 6.34 there is a stagnation of the accuracy according to the reduction dimension.

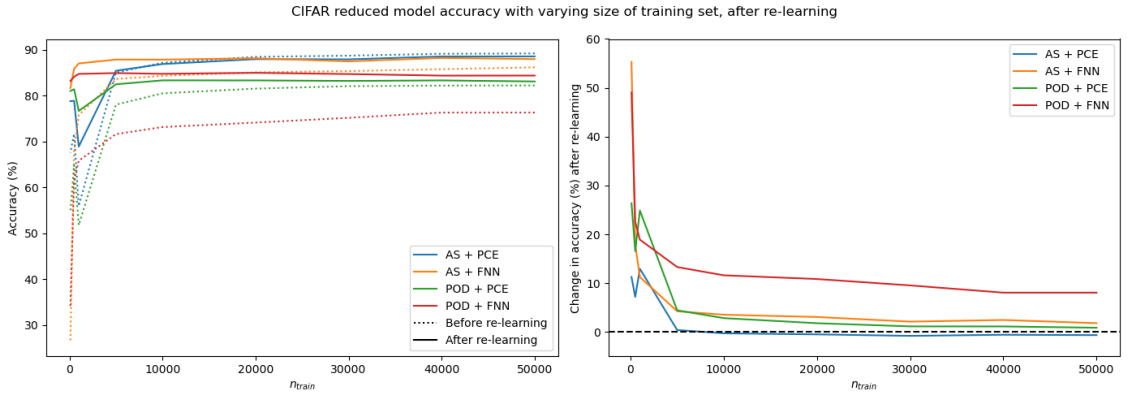


Figure 6.32: CIFAR reduced model accuracy according to amount of training samples, after re-learning.

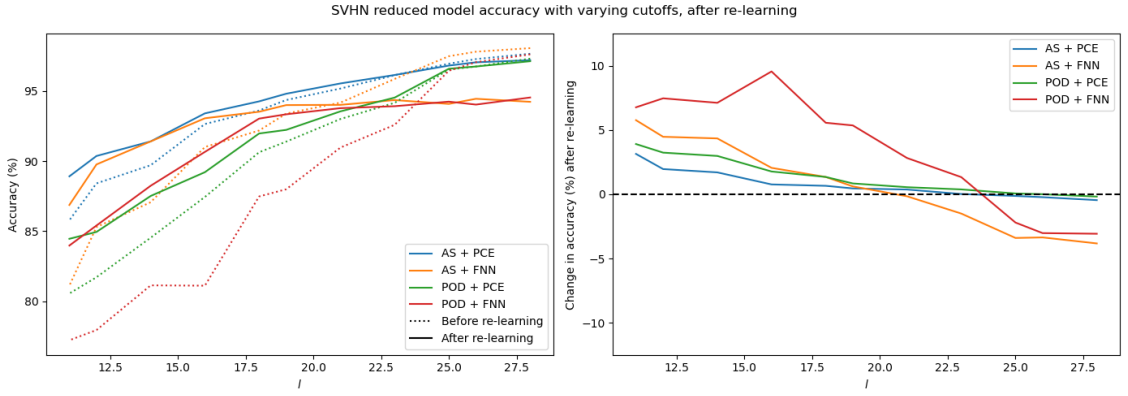


Figure 6.33: SVHN reduced model accuracy according to cutoff, after re-learning.

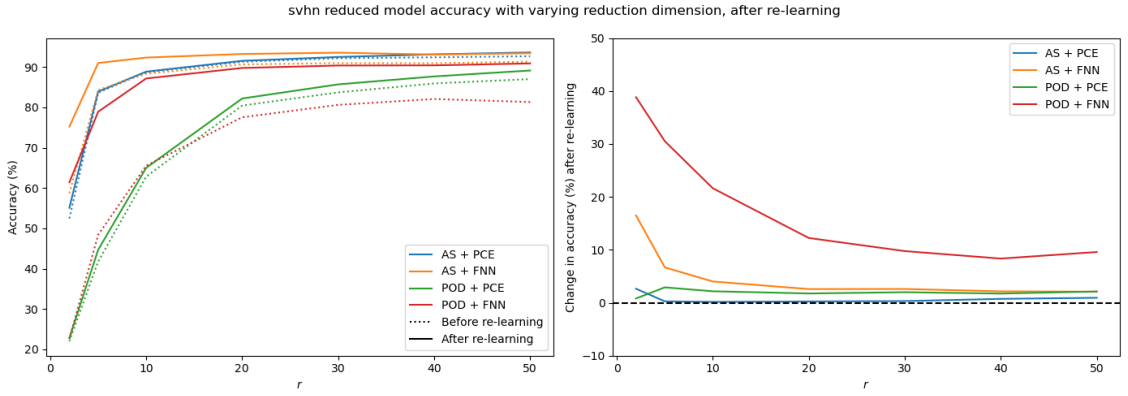


Figure 6.34: SVHN reduced model accuracy according to reduction dimension, after re-learning.

As a contrast to the accuracy curves before re-learning, it seems that the accuracy after re-learning according to r converges to a high accuracy quicker. This also holds for figures 6.27 and 6.24, suggesting that the usage of re-learning allows us to use smaller parameters whilst obtaining the same accuracy as for a higher parameter. There is of course a trade-off: 10 epochs of re-learning for one reduced model took approximately an hour.

Ultimately, it is shown that re-learning will, in most cases, efficiently improve the accuracy.

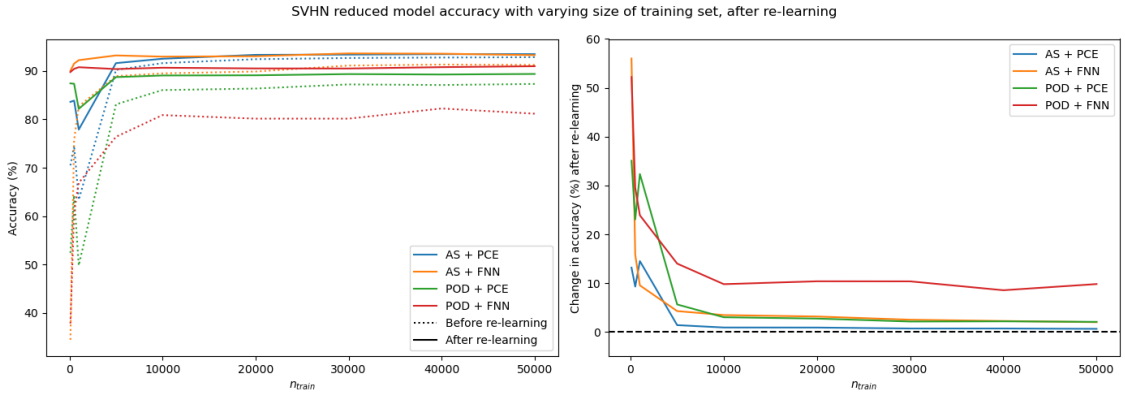


Figure 6.35: SVHN reduced model accuracy according to amount of training samples, after re-learning.

6.4 Qualitative results

In order to give a greater picture of the parameters affecting the performance of a reduced model, a smaller subset of the possible parameters were chosen. From the quantitative results, these seem to give a big variance in the performance of the reduced model whilst simultaneously not severely distorting the results. The aim of the qualitative results is to visualize the effects of tweaking the parameters.

The parameters that will be varied are:

1. The reduced variable r,
2. The cutoff layer l,
3. The amount of samples n_{train} for constructing the projection matrix.

There are obviously other parameters that can be varied as well, such as the learning rate of the FNN, the amount of epochs for the FNN etc. However, the amount of samples, cutoff layer, and reduced variable are known to directly affect both accuracy and computational complexity of the reduced model, and are applicable to all the different combinations available for the reduced model

Such as in section 6.3, the parameters that we keep constant as a base model are $n_{train} = 50000$, $r = 50$, $l = 16$.

The shifts in parameters that will be explored are

- $n_{train} = 5000$,
- $r = 10$,
- $l = 11$.

The shifts in parameters will happen on an isolated basis, i.e. the combination between them will not be explored.

The various combinations are denoted in 3, where the letter given to the combination denotes the variable that is shifted in the respective reduced model, and B denotes the baseline model.

6.4.1 Before re-learning

Table 4 shows the accuracy before conducting re-training on the models. The highest reduced model accuracy for each data set is in **bold font**. It is easy to observe that the combination of AS and PCE yields the highest accuracy in almost all cases.

Combination	n_{train}	r	l
B	50 000	50	16
L	50 000	50	11
R	50 000	10	16
N	5000	50	16

Table 3: Combinations of parameters for the qualitative study.

Data set	Combination	AS+PCE	AS+FNN	POD+PCE	POD+FNN
CIFAR	**B**	**0.8895**	0.8597	0.8231	0.7651
	L	**0.8415**	0.7854	0.7753	0.7116
	R	0.8327	**0.8336**	0.5951	0.6033
	N	**0.8501**	0.8360	0.7802	0.7160
SVHN	**B**	**0.9265**	0.9101	0.8744	0.8111
	L	**0.8578**	0.8112	0.8055	0.7722
	R	**0.8870**	0.8834	0.6285	0.6655
	N	**0.9021**	0.8892	0.8309	0.76

Table 4: Accuracy of the qualitative combinations, before re-learning.

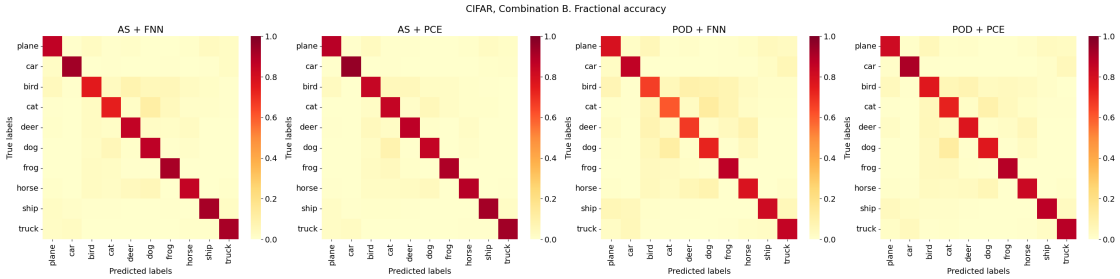


Figure 6.36: Confusion matrices for the reduced models, combination B. (CIFAR)

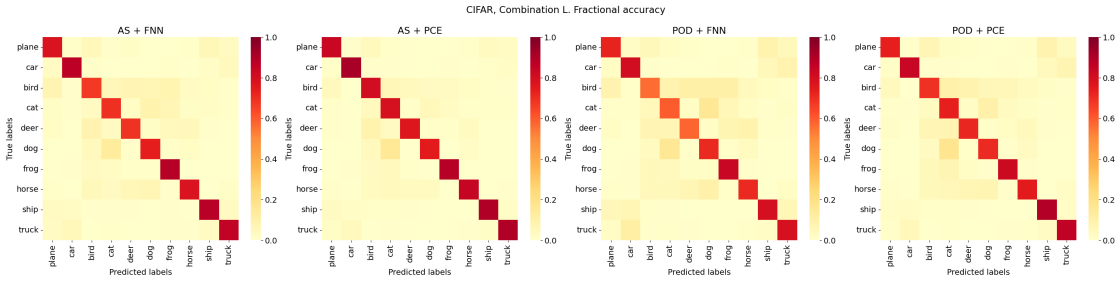


Figure 6.37: Confusion matrices for the reduced models, combination L. (CIFAR)

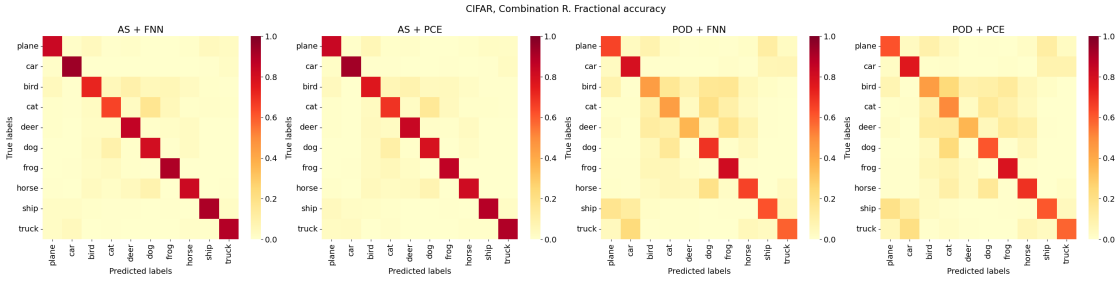


Figure 6.38: Confusion matrices for the reduced models, combination R. (CIFAR)

Figures 6.36, 6.37, 6.39, 6.38, 6.40, 6.41, 6.43, and 6.42 visualize the of the confusion matrices for the output of the qualitative combinations. These can be connected row-wise to the accuracies in

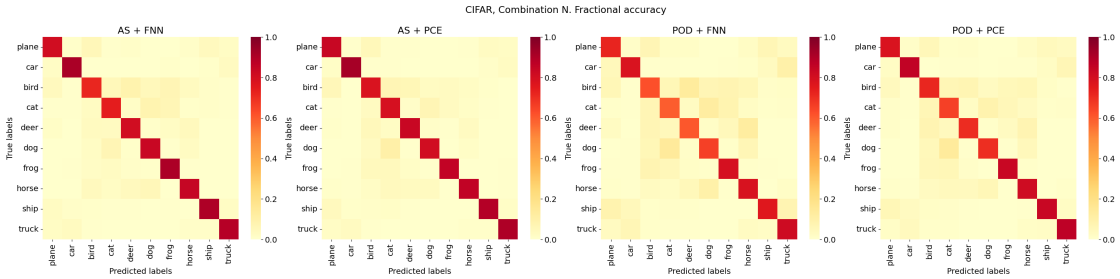


Figure 6.39: Confusion matrices for the reduced models, combination N. (CIFAR)

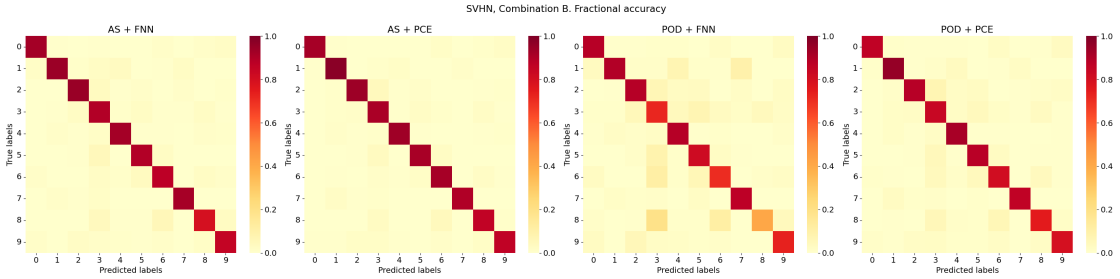


Figure 6.40: Confusion matrices for the reduced models, combination B. (SVHN)

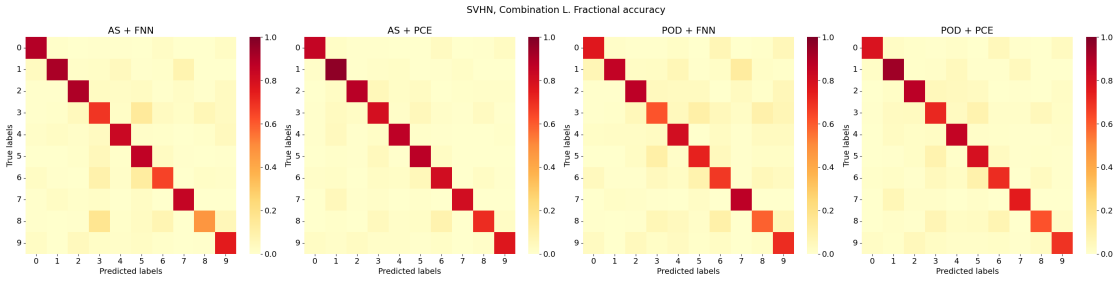


Figure 6.41: Confusion matrices for the reduced models, combination L. (SVHN)

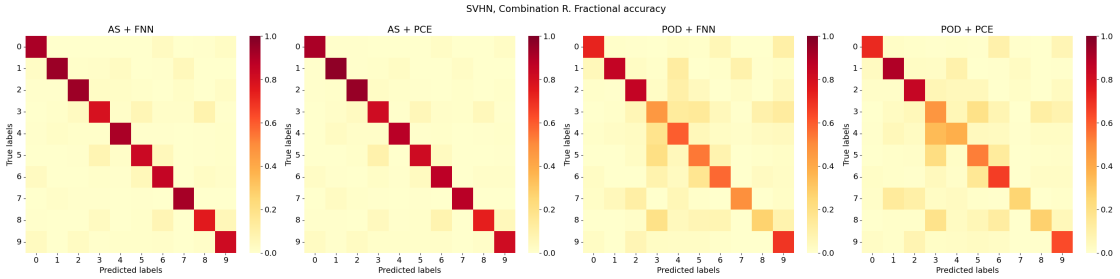


Figure 6.42: Confusion matrices for the reduced models, combination R. (SVHN)

table 4. The numerical value of the per-class-accuracies have been omitted from the plots, however the color bar to the right give an indication of the accuracy of the predictions in the confusion matrix.

The general trend for all combinations is that the diagonal element is quite strong, which is desirable as this indicates that pictures are generally classified correctly. Figure 6.42 and 6.38 feature the highest disruptions of the diagonal pattern, with the POD + PCE and POD + FNN combinations performing poorly. This suggests that using POD in combination with a low reduction dimension r may not be viable. This could indicate that the order of the directions found in AS, that $x^{(l)}$ is projected onto, may be more accurate than the order of the directions found in POD. There does not seem to be a large difference between the confusion matrices obtained from the baseline

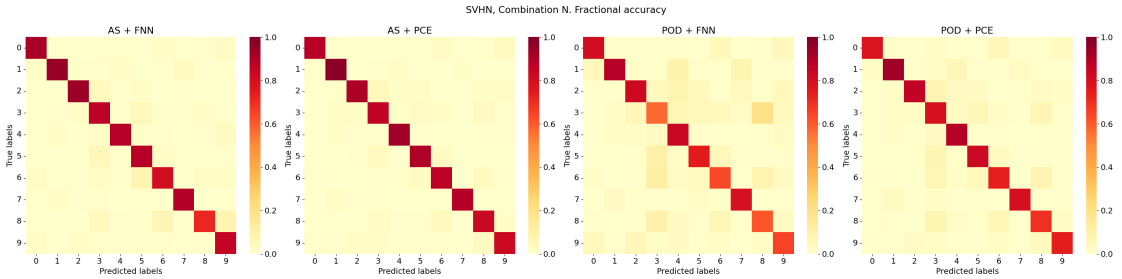


Figure 6.43: Confusion matrices for the reduced models, combination N. (SVHN)

in figure 6.40 and 6.36, indicating that for a larger r the matrices are projecting onto the same subspace. Therefore, the ordering of directions according to the method may be the critical factor for the discrepancies of POD.

On a bright note, when looking closer at the confusion matrices one can see that the predictions with a high false classification rate are between classes where the misclassification is explainable. For example, in figure 6.36 the largest misclassifications are between objects of similar subclasses: cars and trucks, ships and planes, dogs and cats, dogs and horses. Digits that are similar, such as 3 with 5 and 3 with 8, are also the victims of misclassification. There are also misclassifications that appear to be random - for example between 4 and 3 - which may only be explained by poor performance of the model.

Several of the confusion matrices are also having a less strong score on the diagonal for certain elements, indicating that these may be more challenging categories. For example, in figure 6.41, 6.41 and 6.43 there is a lower accuracy score on element 8 and 4 on the diagonal for several of the methods, indicating that these categories may be more ambiguous or too similar to the rest of the classes. The same can be seen in figure 6.36, 6.37, 6.39 and 6.38, where almost all the confusion matrices are having a slightly lower score on the diagonal elements belonging to cat and dog, which are also classes that often are mixed up even in the full model.

Other than the strong diagonal elements in all confusion matrices with some specific exceptions, and the discrepancy in the patterns for combinations using POD with a low r value, there are not any immediately apparent trends. There are some non-diagonal elements of low-medium strength for several of the confusion matrices, however not noticeable enough to comment on them specifically and not showing any clear pattern according to the respective reduced model.

6.4.2 After re-learning

The re-learning process was done with the same hyperparameters for all inputs, as described in section 5. This was to see the effect of knowledge distillation on the different reduced models after 10 epochs, in order to compare them with one another.

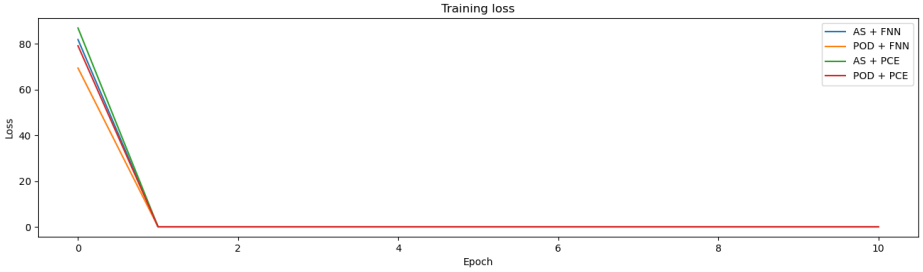


Figure 6.44: The general shape of the training losses for the qualitative combinations.

The training loss curves looked generic for all the combinations listed, with the initial training loss value being quite high, such as in figure 6.44. In epoch 1 the training loss had decreased to some

values around 10^{-5}, where it stayed for the rest of the epochs. This also indicates that there is perhaps no need to train the reduced neural network for more than one or two epochs.

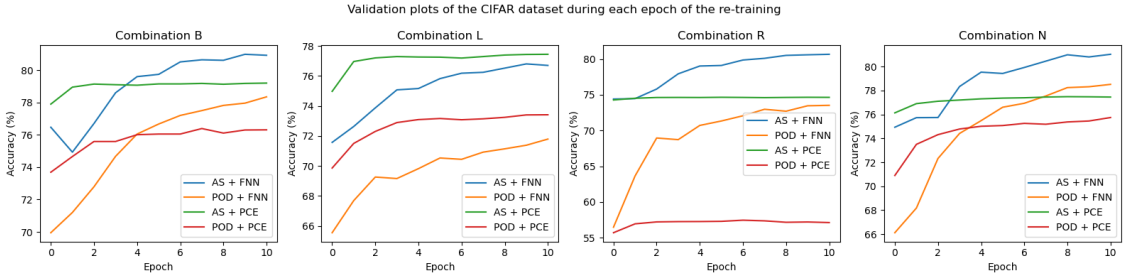


Figure 6.45: The accuracy of the CIFAR models at each epoch of the re-training.

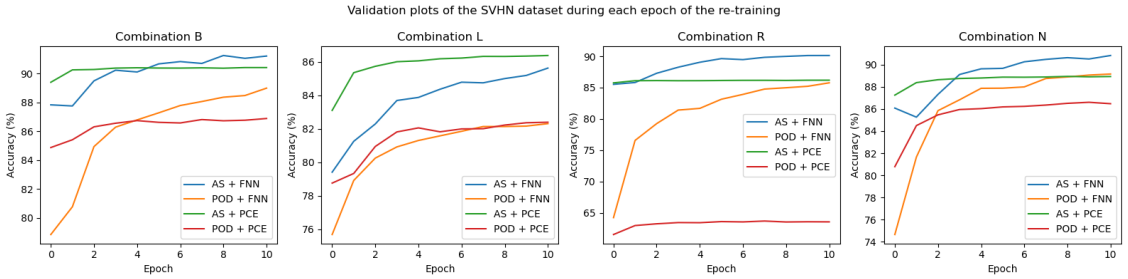


Figure 6.46: The accuracy of the SVHN models at each epoch of the re-training.

From figure 6.45 and figure 6.46 we see the validation of each of the reduced models at every epoch, belonging to the various combinations. In general it is easy to see some trends that apply to both data sets. Firstly, the reduced models using PCE as the input-output mapping are not particularly improving their accuracy during the re-learning process. It is clear to see that, in all cases, the reduced model using POD as a reduction layer and FNN as an input-output mapping is the one gaining the most accuracy during re-learning. However, this is also the combination that starts off with the lowest accuracy, and when looking at the final epoch (10) we can see that all the models converge to around the same accuracy.

For both figures, the combination of POD with PCE generally has the lowest overall accuracy for most parameter combinations, showing that this is a model combination with parameters that are hard to train. It does neither start off with a large accuracy nor improve significantly during the re-training processes. This could also indicate that perhaps the hyperparameters should have been larger for the re-training of particular combination, and can perhaps be attributed to large parameter values in the various layers, so that the small learning rate had little significance during the 10 epochs of re-training.

Data set	Combination	AS+PCE	AS+FNN	POD+PCE	POD+FNN
CIFAR	**B**	***0.8832***	0.8785	0.8337	0.8441
	L	**0.8523**	0.8311	0.7944	0.7734
	R	*0.8279*	**0.8698**	0.6041	0.7837
	N	0.8540	**0.8784**	0.8245	0.8489
SVHN	**B**	**0.9341**	0.9306	0.8921	0.9066
	L	**0.8892**	0.8687	0.8446	0.8397
	R	0.8886	**0.9236**	0.6501	0.8717
	N	0.9161	**0.9320**	0.8872	0.9038

Table 5: Accuracy of the qualitative combinations, after re-learning.

In table 5 we see the accuracy after the re-learning step. Any accuracy that has decreased during the re-training process is in *italic font*. The highest accuracy for each combination is in **bold font.** It can be observed that the accuracy of most of the combinations increases after kd-learning. This

coincides with the results from the quantitative results, where the reduced models with a cutoff before $l = 18$ generally increased its accuracy through re-training.

One may observe in table 5 that the highest accuracy for each combination is equally spread between AS + PCE and AS + FNN. This suggests that AS is a more accurate method for computing the projection method. We can further observe that AS + PCE seems to give a high accuracy for the baseline and combinations with an earlier cutoff-layer, whilst AS + FNN seems to give a higher accuracy for combinations with a lower reduction dimension r or fewer training samples. However, it should be noted that the differences here are minuscule and could vary according to hyperparameters and data sets used.

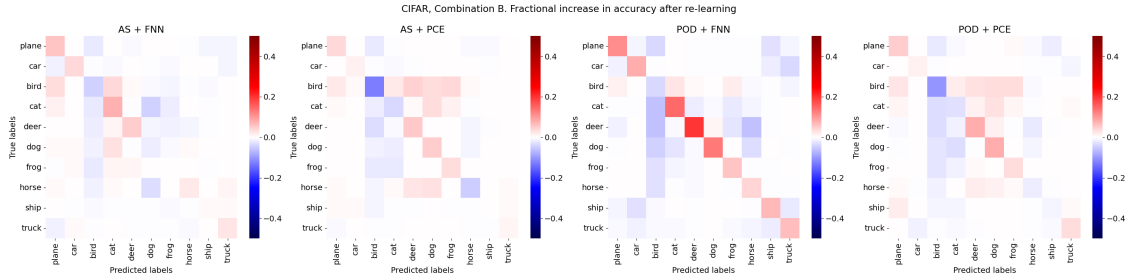


Figure 6.47: The fractional change in accuracy after re-training, for reduced models with Combination B (CIFAR).

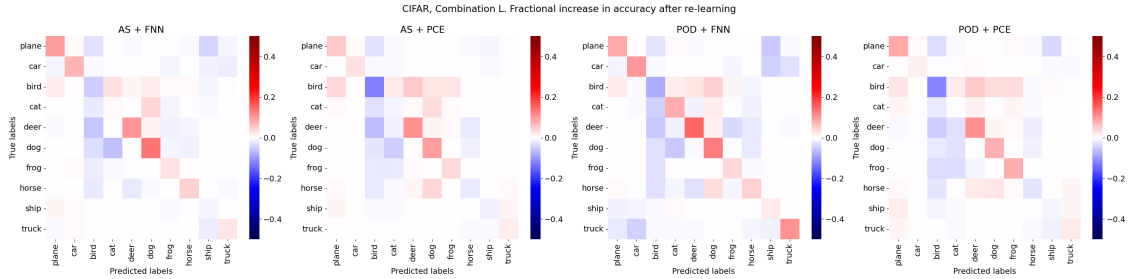


Figure 6.48: The fractional change in accuracy after re-training, for reduced models with Combination L (CIFAR).

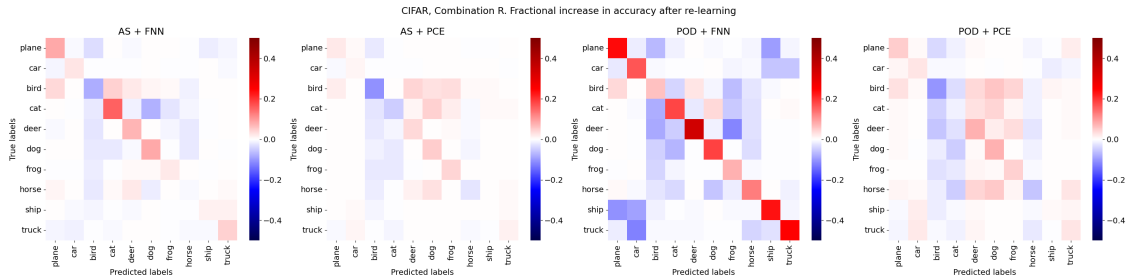


Figure 6.49: The fractional change in accuracy after re-training, for reduced models with Combination R (CIFAR).

Figures 6.47-6.54 show the fractional changes in the confusion matrices after re-learning. The aim of the plots is to get a representative idea of the direct effect of the re-learning on the reduced models. The blue elements in the confusion matrices show that there has been a decrease in the amount of predictions at this particular element. The red elements show an increase in the amount of predictions. Ideally, we want a red diagonal and blue non-diagonals, as this indicates an increase in the correct predictions and a decrease in misclassifications.

A trend of a red diagonal and blue surrounding areas can be viewed in all the figures. Especially for figures 6.53, 6.54, 6.49 and 6.50, we see that the diagonal pattern for the POD + FNN combination

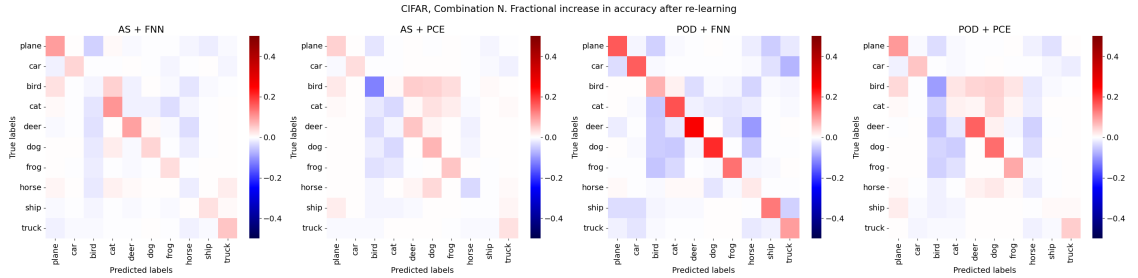


Figure 6.50: The fractional change in accuracy after re-training, for reduced models with Combination N (CIFAR).

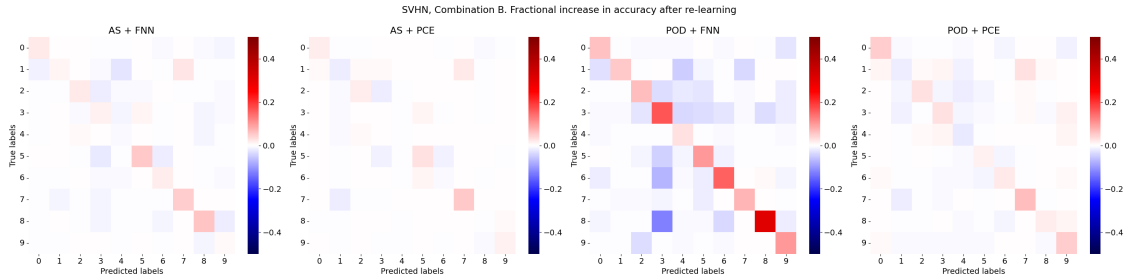


Figure 6.51: The fractional change in accuracy after re-training, for reduced models with Combination B (SVHN).

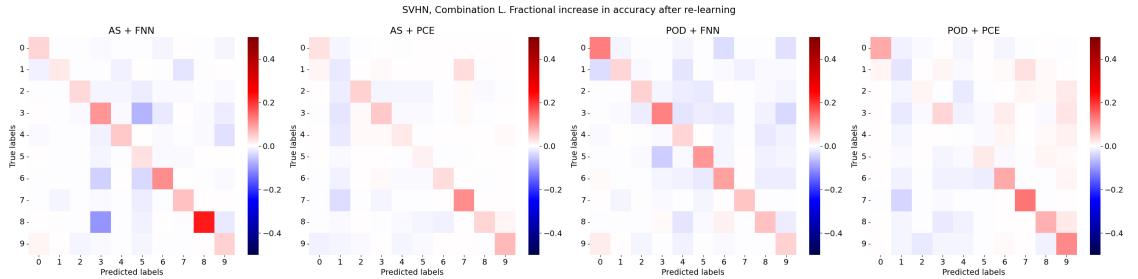


Figure 6.52: The fractional change in accuracy after re-training, for reduced models with Combination L (SVHN).

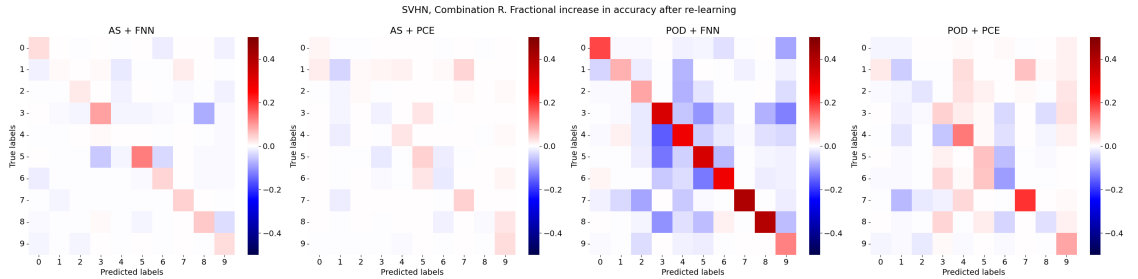


Figure 6.53: The fractional change in accuracy after re-training, for reduced models with Combination R (SVHN).

is both clear and strong, signifying that this is a combination that has improved significantly from the re-training. This correlates with our findings from figures 6.46 and 6.45, where we saw that POD + FNN curves had in general the largest increase in accuracy during re-learning. However, one must recall that these are fractional increases. Although the increase in POD + FNN is tremendous, the accuracy as given in table 5 is still the highest from AS + PCE or AS + FNN.

We can further gather, especially from figures 6.42 and 6.40, that there is very little to the re-

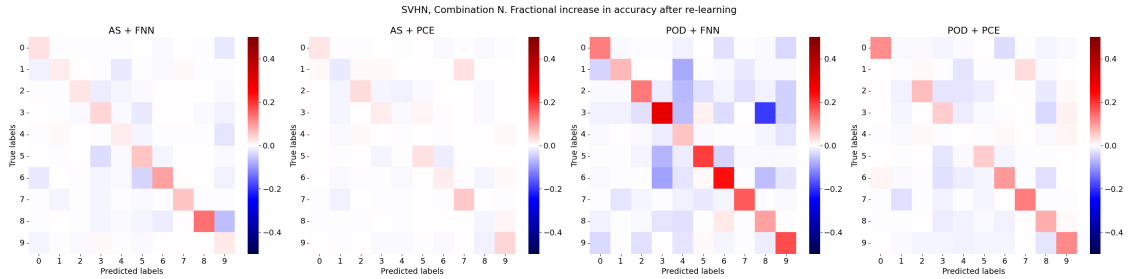


Figure 6.54: The fractional change in accuracy after re-training, for reduced models with Combination N (SVHN).

duced models featuring AS + PCE. This was also found in figures 6.46 and 6.45. The lack of change in the AS + PCE combinations again suggest that we could perhaps have tried with other hyperparameters for the re-training process, or that these combinations already were optimized.

Figures 6.36, 6.37, and 6.38 also reveal the ambiguous categories that we have previously touched upon. We see that there is blue on the diagonal for class 3 i.e. the dog-class, and that generally there is red on the misclassification of dogs as cats and vice versa. This shows negative development in the re-training. During the re-training, the reduced models are training based on the probability distribution of the full model. We have seen that the full model has difficulties with misclassification of cats and dogs, so it is inevitable that the reduced model inherits this trait during the re-training process. Exactly situations like these is why the temperature factor is utilized in the knowledge distillation loss function; to give ambiguity to the outputs of the full model, so that misclassifications are not strongly inherited to the reduced model.

In general, the figures have shown that the re-learning aspect is in the desired direction. Furthermore, that re-learning is more effective and useful for some combinations than others.

6.5 Proximity to solution, using random starting weights

We have seen how model reduction is successful when constructing a reduced model from an already existing model. Although not documented in this thesis, the training and construction of the reduced net are procedures that are time-consuming. The re-learning step with knowledge distillation is shown to be successful for improving models that with low accuracy. A question to pose is whether we can construct a general reduced neural network, and train it with a few epochs to approximate any fully trained neural net.

Using our baseline parameters, we take into consideration the four reduced neural networks that are trained on CIFAR-10, and the four that are trained on SVHN. In both cases they are trained on image classification with 10 classes. The question we pose ourselves is whether they are better at image classification on each other's data sets than a random model (i.e. using their parameters as starting weights are better than using random weights), and whether they can acquire a satisfactory accuracy after an acceptable amount of epochs.

Running the SVHN data set through the CIFAR-trained reduced models, and vice versa, we get the following results:

	Input-output mapping	
Reduction method	FNN	PCE
AS	9.25%	8.28%
POD	9.36%	9.38%

Table 6: Accuracy of the SVHN-trained reduced models on classifying the CIFAR data set, after re-training.

Table 8 is slightly worse than random guessing for a multi-class problem of $n_C = 10$, whilst table 9 is

Reduction method	*Input-output mapping* FNN	PCE
AS	13.87%	13.72%
POD	12.85%	12.97%

Table 7: Accuracy of the CIFAR-trained reduced models on classifying the SVHN data set, after re-training.

performing slightly better than random guessing. Technically this could make sense, as the objects shown in CIFAR are perhaps more complex than SVHN from a human perspective. However, by plotting the confusion matrices as heatmaps we can deem this to be pure coincidence.

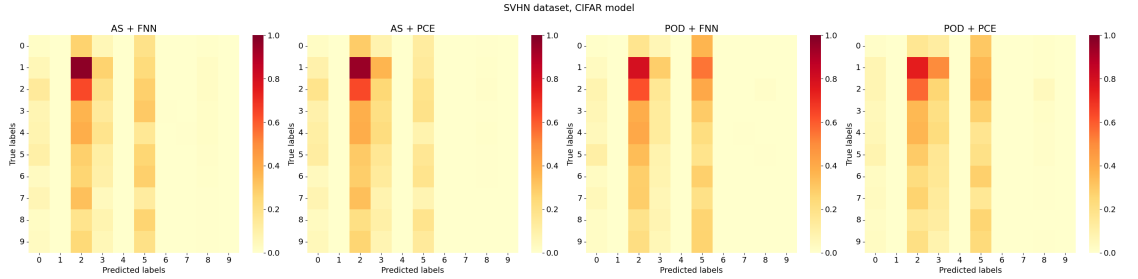


Figure 6.55: Resulting confusion matrices from the classification of the SVHN data set using CIFAR-trained reduced neural networks.

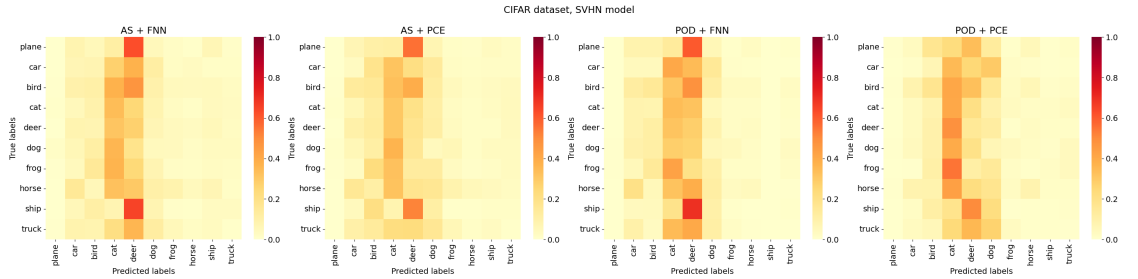


Figure 6.56: Resulting confusion matrices from the classification of the CIFAR data set using SVHN-trained reduced neural networks.

There does not seem to be any clear pattern in figures 6.56 and 6.55. The predictions seem to centered around the middle indexed labels, but this is coincidental.

Following the re-training procedure of the reduced neural network, we ran a knowledge distillation training session of all networks. Note that the hyperparameters used in for the previous re-training processes are not changed, and with the same amount of epochs as well as the same learning rate.

The validation curves in 6.58 and 6.58 immediately show an increase in validation of the experimental models. In both figures, there is a clear order to which reduced models increase the most using re-training: the models using FNN as input-output-mapping are attaining a high accuracy much faster than those using PCE.

All the reduced models start with the same low accuracy. The fact that AS + PCE and POD + PCE does not improve as quick as the FNN-based reduced models suggests what we have previously observed. In general, it seems as if PCE is a method that is poorly poised for the re-training with knowledge distillation using the same hyperparameters as done in the scope of this thesis. The validation curves also suggest that the reason for AS-based reduced models having a high accuracy was solely that it correctly ranked the most important directions. In figure 6.58 and 6.57 it seems purely coincidental the AS + FNN has a higher accuracy than POD + FNN.

Figures 6.60 and 6.59 show the resulting confusion matrices after the re-training for 10 epochs.

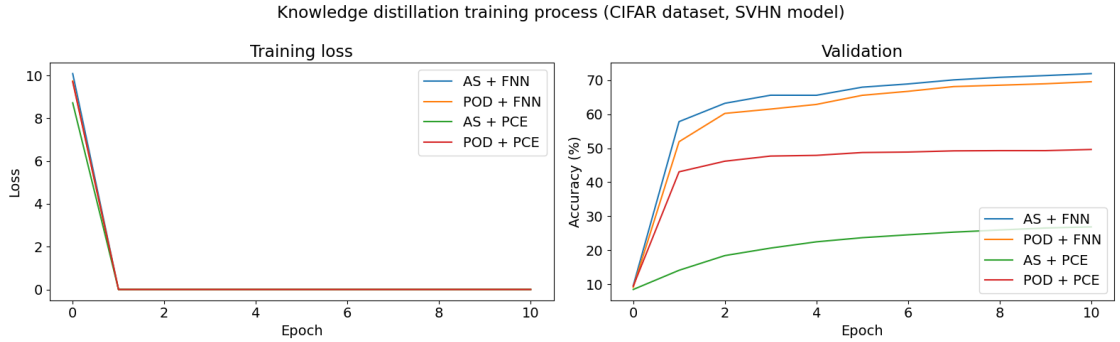


Figure 6.57: Training loss and validation curves for the re-training of the SVHN-trained models, with CIFAR-data.

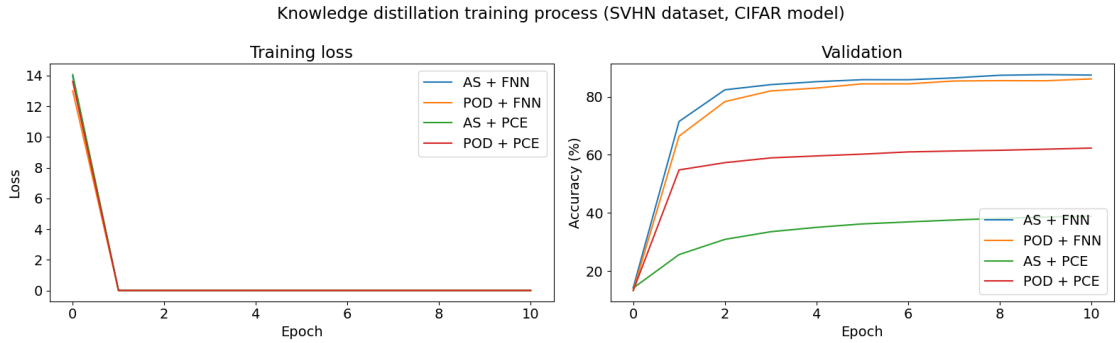


Figure 6.58: Training loss and validation curves for the re-training of the CIFAR-trained models, with SVHN-data.

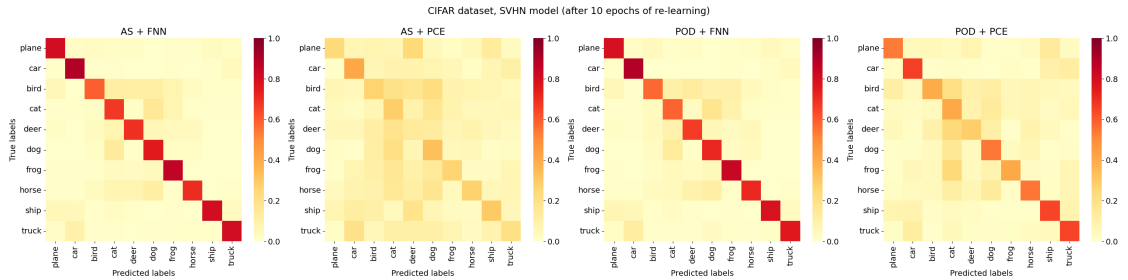


Figure 6.59: Resulting confusion matrices from the classification of the CIFAR data set using SVHN-trained reduced neural networks.

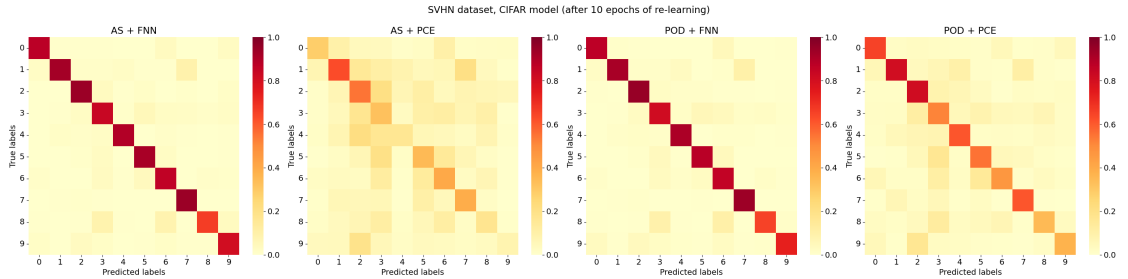


Figure 6.60: Resulting confusion matrices from the classification of the SVHN data set using CIFAR-trained reduced neural networks, after re-training.

The results here coincide with the validation-accuracy curves, and are extremely promising for the notion of a general reduced model The combinations such as AS + FNN and POD + FNN have

Reduction method	*Input-output mapping* FNN	PCE
AS	75.70%	27.17%
POD	73.12%	50.72%

Table 8: Accuracy of the SVHN-trained reduced models on classifying the CIFAR data set, after re-training.

Reduction method	*Input-output mapping* FNN	PCE
AS	88.53%	39.13%
POD	87.35%	62.92%

Table 9: Accuracy of the CIFAR-trained reduced models on classifying the SVHN data set, after re-training.

confusion matrices that are almost equivalent to the reduced models that were carefully reduced to attain for the full models. It also indicates that random initial weights for the pre-model, the projection matrix and the FNN can be an alternative for the AS + FNN method. There are also some pitfalls of these results, of course. AS + PCE and POD + PCE have earlier performed with the highest accuracy, however it seems that re-training PCE with random initial weights does not give desirable results. The accuracy attained from AS + PCE have been higher than the ones seen in tables 9 and 8 for POD + FNN. The hyperparameters, however, have not been tuned specifically for any of the reduced model combinations and so it is unclear whether it is the nature of PCE that is at fault or just the hyperparameters used.

7 Discussion

In section 6, the augmented data of the CIFAR-10 and SVHN data sets were used to train two different models in image classification. The models $\mathcal{ANN}_{CIFAR}$ and $\mathcal{ANN}_{SVHN}$ were both well-performing full models of the class VGG-16. We used the model reduction techniques presented in section 4 and 5 to construct reduced models, obtaining four different combinations for each data set. There were three hyperparameters that were considered to be independent variables subject to change: the cutoff layer l, the reduction dimension r, and the amount of training samples n_{train} used to construct the reduced neural network.

Incremental, qualitative and quantitative results were found from the reduced models. Here are some of the most significant observations:

- Some combinations of projection matrices and input-output mappings are consistently performing better than the rest for any hyperparameter. This can especially be observed before the re-training process. The clear "winner" is the combination of AS + PCE.
- Some combinations are not easily trained during the re-training. This includes all models that have PCE as the input-output-mapping.
- Accuracy is improved by choosing a cutoff layer at a later stage in the model, by choosing a reduction dimension above a certain threshold, and by having a sufficient amount of training samples. It is not necessary to increase the amount of training samples or the reduction dimension excessively to attain a high accuracy. The relationship between the index of the cutoff layer and the accuracy, however, appears to be linear and strictly positive.
- The time spent for calculations is reduced by choosing an early cutoff, and by not using PCE. It seems, when using PCE as an input-output mapping, that there is a positive linear relationship between the calculation time and the reduction dimension r. It also seems that, in general, models using PCE have a greater computation time than models using FNN.
- The storage space taken up by the model is reduced by choosing a late cutoff. It is directly affected by the dimension of the flattened output of the pre-model.
- The re-training using knowledge distillation works very well on reduced models with FNN as the input-output-mapping. It is very unsuccessful for reduced models using PCE.
- In general, reduced models with a very high accuracy perform poorer after re-learning. This may be accredited to over-fitting.
- It is possible and proved successful to use random weights on reduced models, and train them using KD-learning. This seems to be equally successful reducing the models specifically for each data set as presented in 4, for the models using FNN as the input-output layer.

The choice of cutoff, reduction dimension, neural network type, re-learning epochs and so on is clearly a complex issue, as there are trade-offs whenever taking an active choice.

7.1 Computational limitations

Although the reduced model has proven to be successful on the SVHN and CIFAR-10 data sets in terms of accuracy, computation time and so on, there are several limitations to the method. Just the construction of the projection matrices is demanding time-wise and storage-wise. In fact, we were not able to construct it for earlier layers than the ones that were tested in section 6 due to constraints on the Random Access Memory.

It should be emphasized that the data sets used in this study were of size 32×32, which is approximately the size of the icons on the computer screen. In other words, they are abnormally small.

When extending the model reduction method onto data sets of larger sizes, we will run into problems relatively early. In fact, the first months of this thesis were spent trying to perform model reduction on 224×224 size images. Even with High-performance computing, this was in general unsuccessful due to the dimensions simply being too big, and especially in the stage of constructing the projection matrix. Not only does one need to allocate space for the large pre-model output, but taking the SVD of such a huge matrix is extremely computationally expensive.

One could get around this issue by using sketching methods such as Frequent Directions method to construct the projection matrix. This is already implemented within the Active Subspaces library in python (ATHENA), for calculating an approximation to the eigenvalue decomposition. If this method is adequate for AS, it must surely be adequate for POD. In both [34] and [8], it seems that randomized SVD was implemented to attain a truncated SVD. This is also a viable option in order to save both computation time and storage space.

7.2 Tuning of hyperparameters

When introducing the various results in section 6, it was often noted how the hyperparameters were set for all the different models. Furthermore, there was no specific hyperparameter search implemented. This was done in order to have a fair comparison between the results attained, and also due to the thesis being a proof of concept. However, this also restrains the thesis as it excludes the possibility of seeing the extent of accuracy a reduced neural network can provide.

The output of the reduced neural networks have been compared with the output of the full neural networks for the scope of this thesis. Therefore, the accuracies actually serve as replication rates. If the full model output is 90% accurate and the replication rate of the reduced neural network is 90%, it would mean that the reduced model accuracy is actually 81% with regards to the real labels. Although this may be an adequate accuracy, it means that the accuracies provided for the scope of this thesis are much lower with regards to the true labels of the data set.

A possible future research project could be to try to entirely replicate the function of a full neural network, using a reduced neural network. This would require extensive hyperparameter searches for both the construction process and the re-training process of the reduced neural network. Some hyperparameters that would be interesting to look into would be in general those related to the re-training process, such as the optimizer, the temperature factor, the learning rates, batch sizes, and the amount of epochs. Optimizing the parameters during the reduced model construction, such as the choice of cutoff, the reduction dimension, and the amount of training samples can also be beneficial. Already we have seen the relationship between the accuracy and these parameters independently, however different combinations should also be explored and given a numerical accuracy through parameter searches.

7.3 Extension to a more general model

The process of obtaining a reduced model output consisted of training a full model, performing model order reduction of the pre-model output, and either training a FNN or obtaining the coefficients to a PCE model. The time usage of the construction of the reduced model was not documented due to a varying usage of CPUs, which would lead to unfair comparisons. However, this process was approximately equally time demanding to the re-training process. Time usage of re-training was noted in section 5 to be around 1 hour for 10 epochs per model when using 32 CPUs.

In the combination of model order reduction with PCE, we saw that the results were of very high accuracy before the re-training process. There did not seem to be any need for knowledge distillation, as the accuracy did not particularly improve nor had any need for improvement. This was consistent for most combinations that implemented PCE.

In the combination of model order reduction with FNN, the results were generally of lower accuracy before re-training. For these implementations, re-training with knowledge distillation greatly

enhanced the results. This was a necessity for implementations using FNN to attain performance on par with those using PCE.

In section 6.5, we entertained and tested the idea of using random weights on a reduced model. This was to investigate how great of an impact knowledge distillation could have on the reduced model. It was observed that re-training a reduced model using FNN as the input-output mapping gave very a very high accuracy. This opens the possibility of having reduced neural network classes with randomized initial weights. The weights in this context would mean the weights of the pre-model, the weights of the FNN input-output map, and the values of the projection matrix. One could thereby import the reduced neural network, and train it to mimic a full model using knowledge distillation. This approach looks quite similar to the construction of full neural networks using packages such as torchvision. The benefit of this approach, versus just having a full neural networks, is that it produces a neural network that requires less computational time and storage allocation.

Due to the time required for both construction and re-training of reduced neural networks, we thus propose two different approaches for reduced neural networks:

1. Constructing a model-specific reduced neural network with PCE as the input-output map. No re-training.
2. Importing a general reduced neural network with FNN as the input-output map, and with random weights. Re-training it using knowledge distillation.

This is in line with the motivation of creating a reduced neural network i.e. it reduces the model complexity. Both these methods have almost the same time duration and give models that are of around the same accuracy.

8 Conclusion

This thesis has served as a proof of concept to reduced model implementations. Reduced neural networks have been created for the CIFAR-10 and SVHN data sets. The aim of the reduced neural networks have been to reduce model complexity and processing time. The results have shown clear relationships between the desired achievements and various parameters within the reduced models, both quantitatively and qualitatively. Furthermore, the usage of randomly initialized weights in reduced neural networks has been investigated, where we have seen success with some (but not all) reduced neural networks. We have additionally come with two proposals for further work with reduced neural networks, based on the implementation time and accuracy gained. According to results gained, reduced neural networks using PCE can be constructed from scratch without any re-training, whilst reduced neural networks using FNN can be imported with randomized weights and re-trained according to the chosen. The proposals for further work are in line with the objectives of using reduced neural networks, such that they ensure less complex neural networks as well as reduced computation as well as implementation times. Further work in this area would be to find the optimal hyperparameters for the reduced neural networks, and to investigate data sets with larger images.

Bibliography

[1] Bibb Allen et al. '2020 ACR Data Science Institute Artificial Intelligence Survey'. In: *Journal of the American College of Radiology: JACR* 18.8 (Aug. 2021), pp. 1153–1159. ISSN: 1558-349X. DOI: 10.1016/j.jacr.2021.04.002.

[2] 'Best-Fit Subspaces and Singular Value Decomposition (SVD)'. In: *Foundations of Data Science*. Ed. by Avrim Blum, John Hopcroft and Ravindran Kannan. Cambridge: Cambridge University Press, 2020, pp. 29–61. ISBN: 978-1-108-48506-7. DOI: 10.1017/9781108755528.003. URL: https://www.cambridge.org/core/books/foundations-of-data-science/bestfit-subspaces-and-singular-value-decomposition-svd/5FFF3B0EA6B175EC82E64BF4A0AC54E3 (visited on 21st June 2023).

[3] Avrim Blum, John Hopcroft and Ravindran Kannan. *Foundations of Data Science*. Cambridge: Cambridge University Press, 2020. ISBN: 978-1-108-48506-7.

[4] Liv Breivik. 'Dimensional reduction using the snapshot-technique for dynamical systems'. Unpublished. Unpublished.

[5] Linfield C. Brown and Paul Mac Berthouex. *Statistics for Environmental Engineers, Second Edition*. Google-Books-ID: png6zsomkVQC. CRC Press, 29th Jan. 2002. 586 pp. ISBN: 978-1-4200-5663-1.

[6] Yu Cheng et al. 'Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges'. In: *IEEE Signal Processing Magazine* 35.1 (Jan. 2018). Conference Name: IEEE Signal Processing Magazine, pp. 126–136. ISSN: 1558-0792. DOI: 10.1109/MSP.2017.2765695.

[7] Anjir Chowdhury et al. 'A Comparative Study of Hyperparameter Optimization Techniques for Deep Learning'. In: 1st Jan. 2022, pp. 509–521. ISBN: 978-981-19033-1-1. DOI: 10.1007/978-981-19-0332-8_38.

[8] chunfengc. *ASNet*. original-date: 2019-10-10T05:38:14Z. 12th Nov. 2022. URL: https://github.com/chunfengc/ASNet (visited on 24th June 2023).

[9] *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/~kriz/cifar.html (visited on 21st June 2023).

[10] Ray W. Clough. *The Finite Element Method in Plane Stress Analysis*. Google-Books-ID: rfwFHQAACAAJ. American Society of Civil Engineers, 1960. 35 pp.

[11] Paul Constantine et al. 'Python Active-subspaces Utility Library'. In: *The Journal of Open Source Software* 1.5 (29th Sept. 2016), p. 79. ISSN: 2475-9066. DOI: 10.21105/joss.00079. URL: http://joss.theoj.org/papers/10.21105/joss.00079 (visited on 21st June 2023).

[12] Paul G. Constantine, Eric Dow and Qiqi Wang. 'Active Subspace Methods in Theory and Practice: Applications to Kriging Surfaces'. In: *SIAM Journal on Scientific Computing* 36.4 (2014). _eprint: https://doi.org/10.1137/130916138, A1500–A1524. DOI: 10.1137/130916138. URL: https://doi.org/10.1137/130916138.

[13] Chunfeng Cui et al. 'Active Subspace of Neural Networks: Structural Analysis and Universal Attacks'. In: *SIAM Journal on Mathematics of Data Science* 2.4 (Jan. 2020). Publisher: Society for Industrial and Applied Mathematics, pp. 1096–1122. DOI: 10.1137/19M1296070. URL: https://epubs.siam.org/doi/10.1137/19M1296070 (visited on 22nd June 2023).

[14] DeepLearningAI. *C4W1L11 Why Convolutions*. 7th Nov. 2017. URL: https://www.youtube.com/watch?v=ay3zYUeuyhU (visited on 21st June 2023).

[15] Li Deng. 'The mnist database of handwritten digit images for machine learning research'. In: *IEEE Signal Processing Magazine* 29.6 (2012). Publisher: IEEE, pp. 141–142.

[16] Jerry Dischler. *Putting machine learning into the hands of every advertiser - Google Ads Help*. 10th July 2018. URL: https://support.google.com/google-ads/answer/9065075?hl=en (visited on 1st July 2023).

[17] Karl Pearson F.R.S. 'LIII. On lines and planes of closest fit to systems of points in space'. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14786440109462720, pp. 559–572. DOI: 10.1080/14786440109462720. URL: https://doi.org/10.1080/14786440109462720.

[18] Mina Ghashami et al. *Frequent Directions : Simple and Deterministic Matrix Sketching*. 21st Apr. 2015. arXiv: 1501.01711[cs]. URL: http://arxiv.org/abs/1501.01711 (visited on 22nd June 2023).

[19] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[20] Jianping Gou et al. 'Knowledge Distillation: A Survey'. In: *International Journal of Computer Vision* 129.6 (1st June 2021), pp. 1789–1819. ISSN: 1573-1405. DOI: 10.1007/s11263-021-01453-z. URL: https://doi.org/10.1007/s11263-021-01453-z (visited on 22nd June 2023).

[21] David Hartman and Lalit Mestha. 'A deep learning framework for model reduction of dynamical systems'. In: 1st Aug. 2017, pp. 1917–1922. DOI: 10.1109/CCTA.2017.8062736.

[22] Jan S Hesthaven, Gianluigi Rozza and Benjamin Stamm. *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*. SpringerBriefs in Mathematics. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-22469-5 978-3-319-22470-1. DOI: 10.1007/978-3-319-22470-1. URL: https://link.springer.com/10.1007/978-3-319-22470-1 (visited on 21st June 2023).

[23] Geoffrey Hinton, Oriol Vinyals and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 9th Mar. 2015. arXiv: 1503.02531[cs,stat]. URL: http://arxiv.org/abs/1503.02531 (visited on 22nd June 2023).

[24] Wendyam Eric Lionel Ilboudo, Taisuke Kobayashi and Kenji Sugimoto. *TAdam: A Robust Stochastic Gradient Optimizer*. 2nd Mar. 2020. DOI: 10.48550/arXiv.2003.00179. arXiv: 2003.00179[cs,stat]. URL: http://arxiv.org/abs/2003.00179 (visited on 22nd June 2023).

[25] John Davis Jakeman et al. *Polynomial chaos expansions for dependent random variables*. SAND-2019-5228R. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States); Univ. of Stuttgart (Germany); Univ. of Utah, Salt Lake City, UT (United States), 1st June 2019. DOI: 10.2172/1762354. URL: https://www.osti.gov/biblio/1762354 (visited on 21st June 2023).

[26] Kian Katanforoosh, Daniel Kunin and Jiaju Ma. *AI Notes: Parameter optimization in neural networks*. deeplearning.ai. URL: https://www.deeplearning.ai/ai-notes/optimization/ (visited on 20th June 2023).

[27] Ayoosh Kathuria. *Intro to optimization in deep learning: Momentum, RMSProp and Adam*. Paperspace Blog. 13th June 2018. URL: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/ (visited on 21st June 2023).

[28] Roelof Koekoek and René F. Swarttouw. *The Askey-scheme of hypergeometric orthogonal polynomials and its q-analogue*. 19th Feb. 1996. arXiv: math/9602214. URL: http://arxiv.org/abs/math/9602214 (visited on 21st June 2023).

[29] Alex Krizhevsky. 'Learning Multiple Layers of Features from Tiny Images'. In: ().

[30] Edo Liberty. 'Simple and deterministic matrix sketching'. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD' 13: The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Chicago Illinois USA: ACM, 11th Aug. 2013, pp. 581–588. ISBN: 978-1-4503-2174-7. DOI: 10.1145/2487575.2487623. URL: https://dl.acm.org/doi/10.1145/2487575.2487623 (visited on 22nd June 2023).

[31] Cristiano Lima. 'Analysis — The AI debate is sweeping through the federal government'. In: *Washington Post* (21st June 2023). ISSN: 0190-8286. URL: https://www.washingtonpost.com/politics/2023/06/21/ai-debate-is-sweeping-through-federal-government/ (visited on 1st July 2023).

[32] Breivik Liv. *Combining artificial neural networks with reduced order models with applications to classificaiton problems, Github repository*. URL: https://github.com/livbrvk/RedMods (visited on 5th July 2023).

[33] Laurens van der Maaten, Eric Postma and Jaap van der Herik. 'Dimensionality Reduction: A Comparative Review'. In: ().

[34] *mathLab/Smithers*. original-date: 2021-04-09T09:06:15Z. 27th Mar. 2023. URL: https://github.com/mathLab/Smithers/blob/7199dbd531487ec35e843a8f5971b7f3061db4e3/README.md (visited on 22nd June 2023).

[35] Warren S. McCulloch and Walter Pitts. 'A logical calculus of the ideas immanent in nervous activity'. In: *The bulletin of mathematical biophysics* 5.4 (1st Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 1st July 2023).

[36] Laura Meneghetti. *A Reduced Order Approach for Artificial Neural Networks applied to Object Recognition*. 2022. URL: https://hdl.handle.net/20.500.11767/129570 (visited on 20th June 2023).

[37] Laura Meneghetti, Nicola Demo and Gianluigi Rozza. *A Proper Orthogonal Decomposition approach for parameters reduction of Single Shot Detector networks*. 27th July 2022. DOI: 10.48550/arXiv.2207.13551. arXiv: 2207.13551[cs,math]. URL: http://arxiv.org/abs/2207.13551 (visited on 20th June 2023).

[38] Yuval Netzer et al. 'Reading Digits in Natural Images with Unsupervised Feature Learning'. In: ().

[39] Michael A. Nielsen. 'Neural Networks and Deep Learning'. In: (2015). Publisher: Determination Press. URL: http://neuralnetworksanddeeplearning.com (visited on 20th June 2023).

[40] OpenAI. *Introducing ChatGPT*. 22nd Nov. 2022. URL: https://openai.com/blog/chatgpt (visited on 1st July 2023).

[41] Jeff M. Phillips. 'Big Data and Sketching'. In: *Mathematical Foundations for Data Analysis*. Ed. by Jeff M. Phillips. Springer Series in the Data Sciences. Cham: Springer International Publishing, 2021, pp. 261–281. ISBN: 978-3-030-62341-8. DOI: 10.1007/978-3-030-62341-8_11. URL: https://doi.org/10.1007/978-3-030-62341-8_11 (visited on 22nd June 2023).

[42] *PyTorch*. URL: https://www.pytorch.org (visited on 21st June 2023).

[43] Srikumar Ramalingam. 'How the backpropagation algorithm works'. In: ().

[44] Francesco Romor, Marco Tezzele and Gianluigi Rozza. 'ATHENA: Advanced Techniques for High dimensional parameter spaces to Enhance Numerical Analysis'. In: *Software Impacts* 10 (Nov. 2021), p. 100133. ISSN: 26659638. DOI: 10.1016/j.simpa.2021.100133. URL: https://linkinghub.elsevier.com/retrieve/pii/S2665963821000543 (visited on 25th June 2023).

[45] Gianluigi Rozza et al. 'Basic ideas and tools for projection-based model reduction of parametric partial differential equations'. In: *Volume 2: Snapshot-Based Methods and Algorithms*. Ed. by Peter Benner et al. Berlin, Boston: De Gruyter, 2021, pp. 1–47. ISBN: 978-3-11-067149-0. DOI: doi:10.1515/9783110671490-001. URL: https://doi.org/10.1515/9783110671490-001 (visited on 21st June 2023).

[46] Olga Russakovsky et al. 'ImageNet Large Scale Visual Recognition Challenge'. In: *International Journal of Computer Vision* 115.3 (1st Dec. 2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y (visited on 2nd July 2023).

[47] *scikit-learn: machine learning in Python — scikit-learn 1.3.0 documentation*. URL: https://scikit-learn.org/stable/ (visited on 7th July 2023).

[48] *SciPy documentation — SciPy v1.12.0.dev Manual*. URL: https://scipy.github.io/devdocs/index.html (visited on 7th July 2023).

[49] Agam Shah. *How Tesla Uses and Improves Its AI for Autonomous Driving*. URL: https://www.enterpriseai.news/2023/03/08/how-tesla-uses-and-improves-its-ai-for-autonomous-driving/ (visited on 1st July 2023).

[50] Connor Shorten and Taghi M. Khoshgoftaar. 'A survey on Image Data Augmentation for Deep Learning'. In: *Journal of Big Data* 6.1 (6th July 2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0 (visited on 24th June 2023).

[51] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 10th Apr. 2015. DOI: 10.48550/arXiv.1409.1556. arXiv: 1409.1556[cs]. URL: http://arxiv.org/abs/1409.1556 (visited on 21st June 2023).

[52] Tomasz Szandała. 'Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks'. In: *Bio-inspired Neurocomputing*. Ed. by Akash Kumar Bhoi et al. Vol. 903. Series Title: Studies in Computational Intelligence. Singapore: Springer Singapore, 2021, pp. 203–224. ISBN: 9789811554940 9789811554957. DOI: 10.1007/978-981-15-5495-7_11. URL: http://link.springer.com/10.1007/978-981-15-5495-7_11 (visited on 22nd June 2023).

[53] Marco Tezzele et al. 'An integrated data-driven computational pipeline with model order reduction for industrial and applied mathematics'. In: *Novel Mathematics Inspired by Industrial Challenges*. Ed. by Michael Günther and Wil Schilders. Mathematics in Industry. Cham: Springer International Publishing, 2022, pp. 179–200. ISBN: 978-3-030-96173-2. DOI: 10.1007/978-3-030-96173-2_7. URL: https://doi.org/10.1007/978-3-030-96173-2_7 (visited on 21st June 2023).

[54] *The Street View House Numbers (SVHN) Dataset*. URL: http://ufldl.stanford.edu/housenumbers/ (visited on 21st June 2023).

[55] Neil Thompson et al. 'The Computational Limits of Deep Learning'. In: *Ninth Computing within Limits 2023*. Ninth Computing within Limits 2023. Virtual: LIMITS, 14th June 2023. DOI: 10.21428/bf6fb269.1f033948. URL: https://limits.pubpub.org/pub/wm1lwjce (visited on 1st July 2023).

[56] Alan Turing. 'Intelligent Machinery'. In: 1948.

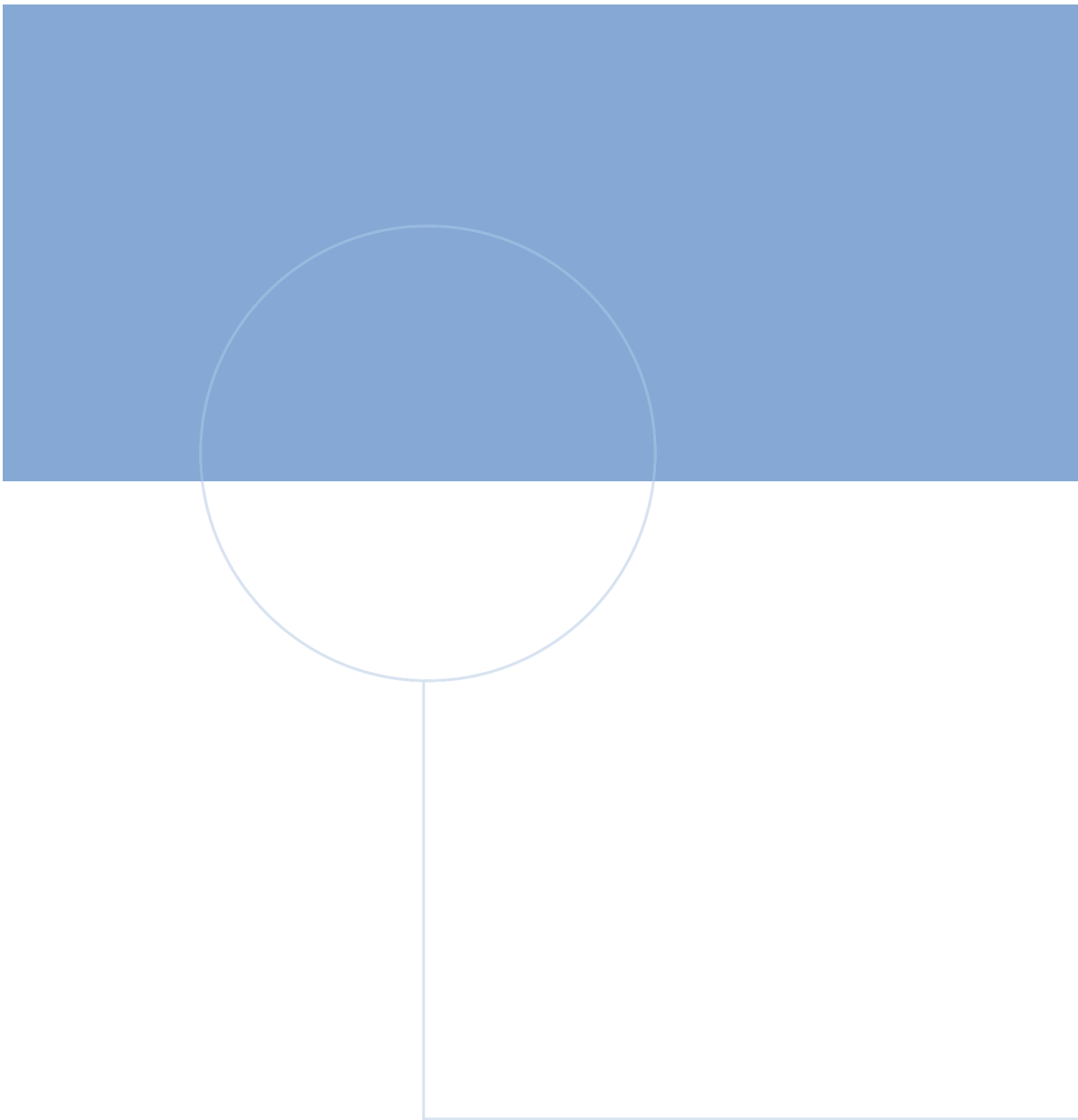
[57] Feng Wang and Huaping Liu. *Understanding the Behaviour of Contrastive Loss*. 20th Mar. 2021. arXiv: 2012.09740[cs]. URL: http://arxiv.org/abs/2012.09740 (visited on 5th July 2023).

[58] L. Wang, G. Libert and P. Manneback. 'Kalman filter algorithm based on singular value decomposition'. In: *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*. [1992] Proceedings of the 31st IEEE Conference on Decision and Control. Dec. 1992, 1224–1229 vol.1. DOI: 10.1109/CDC.1992.371522.

[59] *WebD2: Web Graphics Basics*. URL: https://www.washington.edu/accesscomputing/webd2/student/unit4/module2/web_graphics_basics.html (visited on 21st June 2023).

[60] Matt Weinberg. 'Lecture 11: Low Rank Approximation and the Singular Value Decomposition'. In: ().

[61] Eric W. Weisstein. *Convolution*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ (visited on 21st June 2023).



NTNU

Norwegian University of
Science and Technology