Master's thesis

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

Camilla Balestrand Klemetsen

Neural Networks on Low-Rank and Stiefel Manifolds

Master's thesis in Applied Physics and Mathematics
Supervisor: Elena Celledoni
June 2022

NTNU
Norwegian University of
Science and Technology

Camilla Balestrand Klemetsen

Neural Networks on Low-Rank and Stiefel Manifolds

Master's thesis in Applied Physics and Mathematics
Supervisor: Elena Celledoni
June 2022

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

NTNU
Norwegian University of
Science and Technology

Abstract

In the context of deep learning as optimal control, we investigate the effects of training neural networks on the low-rank and Stiefel manifolds. Low-rank and orthogonality is preserved through numerical geometric integration. In this way, we formulated networks evolving on lower-dimensional manifolds. This is based on the hypothesis that we do not need the entire dimension of the input space to make a classification. We see that Residual Neural Networks (ResNet) performs as well on a truncated singular value decomposition of the data as the original. We show that the developed methods perform as well, or nearly as well, as the original ResNet formulation in terms of accuracy. One of the methods is also an order-reduction method, which has fewer trainable parameters per layer. Furthermore, we show that our methods are more robust in terms of adversarial Fast Gradient Sign attacks, and we argue that this benefit is a result of the structure-preservation during numerical integration.

Sammendrag

I denne oppgaven ser vi på effekten av dyp læring som optimal kontroll på mangfoldigheter. Vi utvikler og trener flere nettverk som bevarer lav-rang og ortogonalitet i treningsprosessen. Bibetingelsen til optimal kontroll problemet er en ordinær differensialligning, og invariantene blir bevart igjennom geometrisk numerisk integrasjon av denne. Vi utviklet disse metodene etter å ha sett at Residuale Nevrale Nettverk (ResNet) klassifiserte like godt med input som var trunkert med singulær verdi dekomposisjon som orginale data. De utviklede metodene klassifiserer like godt, eller nesten like godt, som den orginale ResNet formuleringen. En av methodene er også en redusert-orden metode som har færre trenbare parametre per lag i nettverket. Til slutt, så viser vi at våre metoder er mer robust mot såkalte "Adversarial Attacks", og argumenterer for at dette har sammenheng med bevaring av struktur og i all hovedsak ortogonalitet.

Preface

This thesis is the final project of my master's degree in Industrial Mathematics at the Norwegian University of Science and Technology. The work has been done in close collaboration with my supervisor Elena Celledoni, and I want to thank her for inspiring me though our conversations, and for including me in research group meetings and conferences. I want to thank the participants at MaGiC for giving motivating talks and for nudging me in the right direction.

I also want to give a thanks to my family and friends for the encouragement and always believing in me, not only the past semester, but through all my years studying. Especially to my dad who always knew the struggle I was going through. A special thanks to Matteland and my classmates for the invaluable companionship and joy.

Lastly, I also want to thank myself for coming this far. I would never have believed I would be finishing a masters degree in mathematics ten years ago, and I am immensely proud of myself. But I could never have done this alone.

Bergen, June 22nd 2022,
Camilla

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Youth is easily deceived because it is quick to hope.
– Aristotle

In recent years, we have seen an increased interest in machine learning and neural networks, and for good reason. New research has shown neural networks to be flexible and powerful. We are able to use them on problems from speech recognition and natural language processing. It can also be used to predicting weather patterns, increasing precision in cancer treatments, robots and self-driving cars and many more. Whether or not it is fully justified, deep learning is a topic of interest to the industry [37]. However, from a mathematical point of view, neural networks are not well understood. However, numerical experiments show promising results and it would be naive not to try to understand them as mathematical objects.

We can describe the continuously learning system as a differential equation. We control the learning by tweaking the parameters of the equation. Then, the differential equation is solved by determining the functions satisfying it. This is often called integrating the equation. In this way, we can determine the optimal parameters of the differential equation such that the learning problem is solved. This is deep learning as optimal control.

We can describe the continuously learning system as a differential equation. We control the learning by determining the parameters in the equation solving an optimisation problem. In this way, we can determine the optimal parameters of the differential equation such that the learning problem is solved. This approach is a way of interpreting deep learning as an optimal

control problem [62, 28, 26, 6, 21].

There are many ways to integrate ordinary differential equations (ODEs). When the closed-form analytical solution cannot be found, we resort to numerical solutions. Within numerical integration, structure-preserving algorithms emerged from many different research areas. Areas such as molecular dynamics, mechanics, theoretical physics and other fields of computational science. They required the use of numerical methods preserving, to high accuracy, fundamental geometric properties of the underlying system. These methods showed improved qualitative behaviour compared to traditional general-purpose methods [29]. Therefore, investigating which properties these methods have in the field of machine learning is intriguing.

Artificial intelligence and machine learning algorithms are entering the real world. We see classification algorithms, anomaly detectors, automation algorithms in self-driving cars and many more. As these algorithms enter the real world to relieve humans of tedious tasks we are also, therefore, relying on these algorithms to be robust to changes and malicious attacks. For this reason, a better understanding of these as mathematical objects may help us on the path to more robust models.

In this thesis, we will use geometry to motivate structure preservation in neural networks. In particular, the Manifold Hypothesis. This allows us to formulate methods for conserving orthogonality and low-rank in the training of the neural network. We will conduct experiments and show how the performance does not deteriorate using these methods. Lastly, we also show some promising results on adversarial attacks, where the developed methods perform better than the traditional Residual Neural Network.

This thesis is structured as follows. Chapter 2 presents the Manifold Hypothesis the and motivation behind the thesis. Methods for linear dimensionality reduction are presented and discussed. Chapter 3 gives an introduction to the theory behind deep learning with neural networks for the classification problem. More specifically we introduce the Residual Neural Network which we use as a nonlinear transformation to learn image manifolds. In Chapter 4 we investigate and implement some of the theory and algorithms discussed in the previous chapters on several data sets. We show and compare their performance and test the methods' robustness to adversarial attacks. In Chapter 5 we examine and discuss the numerical results, and provide suggestions for future work. Then we conclude our findings in Chapter 6.

Chapter 2

The Manifold Hypothesis and Reduced-Order Models

Of course this is happening inside your head, Harry,
but why on earth should that mean it is not real?
– *Albus Dumbledore*, Harry Potter and the Deathly Hallows

In this chapter we will introduce the manifold hypothesis. This hypothesis motivates the choice to look for and conserve underlying structures in data sets. In particular, manifold structures. Then we proceed into reduced-order models and illustrate why reduced-order models and the manifold hypothesis share similar goals. This enables us to do a reduced-order coordinate change where the goal is to preserve these structures. The models presented in this chapter will be combined with the neural network in the next chapter.

2.1 Smooth Manifolds and Topology

When working with images or videos, these data types are represented by matrices. The columns of a matrix $A \in \mathcal{S} \subset \mathbb{F}^n$ *span* a (linear) subspace $\mathcal{S}$, and the column rank gives the dimension of $\mathcal{S}$. Thus, the gray-scale image can be though of as a surface $\mathcal{S} \subset \mathbb{R}^{m \times n}$, where the dimension of the image is $(m \times n)$ pixels. Therefore, the set of all $(m \times n)$ matrices forms a vector space. Assume, for a moment, that the collected images are all full rank. Then, this collection of images form a *manifold* [2]. A differentiable manifold is a topological space with local coordinate patches that allow for differentiability [60]. A 1-dimensional manifold is shown in Figure 2.1. The

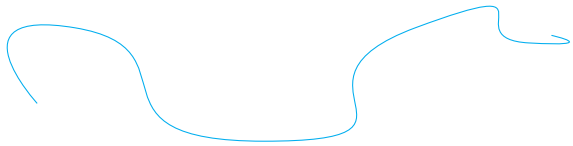


Figure 2.1: 1-dimensional manifold.

blue line has zero volume and area, but the curve lives in the plane $\mathbb{R}^2$. The manifold is locally Euclidean; it is *homeomorphic* to $\mathbb{R}^1$. Furthermore, a 2-dimensional manifold, such as a sphere in $\mathbb{R}^3$, is locally a flat plane. This concept is important as this allows us to work with derivatives and continuous maps in the manifold setting. These local coordinate patches are referred to as *charts*. A chart ϕ on the manifold $\mathcal{M}$ is a bijection of a subset of $\mathcal{M}$ onto an open subset of the Euclidean space $\mathbb{R}^{m\times n} \supset \mathcal{M}$. If we can cover the manifold with charts we form an *atlas*. If the charts making up the atlas are *diffeomorphisms*, then the atlas is smooth and differentiable [60]. The existence of tangents on the manifold is the single most important property of the manifold. Consider the point $p \in \mathcal{M}_k$. For any curve $Y(t)$ such that $p = Y(t_1)$, the tangent vector to $Y'(t)|_{t=t_1}$ is an element of $\mathcal{T}_p\mathcal{M}_k$. For a manifold $\mathcal{M}_k$, the collection of the tangent planes at all points $p \in \mathcal{M}_k$ is the tangent bundle of the manifold and it is a manifold in its own right. Finally, a *(tangent) vector field* on $\mathcal{M}$ is is smooth function $F : \mathcal{M} \to \mathcal{T}\mathcal{M}$ such that $F(p) \in \mathcal{T}_p\mathcal{M}$ for all $p \in \mathcal{M}$ [46]. These tangent vector fields now allows us to discuss differential equations evolving on manifolds, but we will continue this discussion later in the chapter.

Whether the set of *all* images of size $m \times n$ form a manifold is not known. But, we can argue that images of numbers or faces are related [63]. This structured relation is quite far from the set of all matrices, which also contains white noise. Random choices will not generate the structure we see in images of faces. This could be an indication that some classes of images, which have this structure in common, lives in a subset of this higher-dimensional Euclidean space.

2.2 The Manifold Hypothesis

Data is being produced at a fast rate. As technology improves, we make room for data of higher quality. In practice, this results in matrices and vectors of higher dimensions. When studying images or matrices $X \in \mathbb{R}^{m\times n}$, the manifold hypothesis assumes these images naturally lie in a subspace $\mathcal{M} \subset \mathbb{R}^{m\times n}$,

an embedded sub-manifold in this higher-dimensional Euclidean space. If a manifold $\mathcal{M}$ is embedded into $\mathbb{R}^D$ there exists a homeomorphism from $\mathcal{M}$ to $\mathbb{R}^D$ [9]. A high-resolution image is just a high dimensional sample of the underlying image manifold. Therefore, by a certain coordinate change to a chart on the manifold, we need fewer parameters to represent the image-matrix [2]. Indications of the manifold assumption to be true could be observed as the images not being of full rank, i.e. the matrix does not span the whole space or other properties inherent to the matrices. If this is the case for all images in this class, it might be a strong indication for the manifold hypothesis to be true. Finding this underlying structure may help us reduce the complexity at hand, and we can apply more computationally efficient techniques for image classification and so on. The manifold hypothesis can, in some sense, explain why deep learning performs so well at classification problems, making sense of large amounts of high dimensional data. It only needs to focus on key features as the data lives in a subspace of the input space [12].

There seems to be other motivating factors for learning the underlying manifold. If we imagine the data manifold as a surface, where each point is an image, traversing along the manifold and we get another valid sample, as seen in [82]. In [34], they show that we might end up learning unwanted traits if one introduced too much noise into the data set. For instance, there might be a sub-manifold in the animal manifold which contains cats. Moving away from this, i.e. perturbing the image, we find animals that look like cats but are not, or just noisy images.

The manifold hypothesis has a collection of theoretical and experimental indications. Numerous papers have been produced to confirm the manifold assumption using carefully constructed high-dimensional data sets, such as in [19, 4, 70]. In these articles, the authors try to verify whether points on a low dimensional manifold can be projected onto a higher-dimensional manifold. They show that samples from a circle can be projected onto the sphere.

We can argue that related data are close in space, as they share some of the same properties and are therefore, in a sense, *similar*. This is the idea behind clustering techniques such as K-means, where we assume each class $k \in K$ is contained in a separate sub-manifold. Therefore, using the notion of distance from topology and geometry is a natural tool to analyse various kinds of data, as argued in [8]. In this paper, the author argues that we should not restrict our attention to properties dependent on the choice of coordinates, as these are not inherent to the data and do not carry meaning

themselves. Coordinates typically show the data from a point of view and will not show the complete picture. Topology can shed light on this issue, as topology studies geometric properties that are insensitive to metrics such as curvature and thus are coordinate-free.

It is worth noting that the manifold hypothesis is not always true. However, according to the Johnson-Lindenstrauss lemma [48], we can always construct a lower-dimensional representation of our data, which preserves a desired property, like the orthogonal projection. More accurately, a set of points in a high-dimensional space can be embedded into a low dimensional space such that distances are nearly preserved. This means that we can select invariants in the data and embed them into lower-dimensional spaces. So even if the manifold assumption breaks in our case, we could still hope to get some value from our model. We will investigate data sets to see whether we can find indications of a lower-dimensional structure, in particular low-rank and orthogonality. This will allow us to train a network on these manifolds.

2.3 Reduced Order Models

Under the assumption of the manifold hypothesis; manifold learning aims at finding these key parameters to reduce the dimension of the problem. Dimension reduction and manifold learning are interrelated so that solving one leads to the solution of the other [81]. Therefore, it is tempting to use dimension reduction and manifold learning techniques such as Principal Component Analysis (PCA) and multidimensional scaling (MDS), Isomap and Local Linear Embedding (LLE), to name a few, as seen in [9, 15, 74] for manifold learning. In our case, we will focus on dimension reduction methods.

From geometry, if two objects have the same shape but are, i.e. rotated or scaled, we say two objects are *similar*. In Euclidean spaces, a bijection from the space onto itself that scales or rotates points in a fixed ratio is a similarity or a similarity transformation. Such similarities preserve invariants [68]. We are interested in similarity transforms for matrices. The matrix $A \in \mathbb{R}^{n\times n}$ can be expressed as a product of other matrices which share many properties with A, but might have a more useful form.

Definition 2.1 *We say that the matrices $A, B \in \mathbb{R}^{n\times n}$ are similar if there exists an invertible matrix M such that*

$$B = M^{-1}AM. \tag{2.1}$$

The matrix M provides a similarity transformation from A to B. In the case of M being unitary, we say A and B are unitarily equivalent.

Both Schur's lemma and the spectral theorem present similarity transformations for square matrices. In the more general case of rectangular matrices, we need matrices M_1 and M_2 of different sizes to satisfy (2.1). In this case, we encounter the eigenvalue decomposition and the Singular Value Decomposition (SVD), which we will be working with throughout the thesis. We will also consider a continous interpretation of the SVD, the dynamic low-rank approximation.

2.3.1 The Singular Value Decomposition

Let $A \in \mathbb{R}^{m\times n}$, and assume $m \geqslant n$ for simplicity. As A is a rectangular matrix, there exist rectangular and orthogonal matrices, U and V, such that (2.1) is satisfied. The orthogonal matrices $U \in \mathbb{R}^{m\times m}$ where $U^TU = I_m$ and $V \in \mathbb{R}^{n\times n}$ where $V^TV = I_n$ such that

$$A = U\Sigma V^T, \tag{2.2}$$

where $\Sigma \in \mathbb{R}^{m\times n}$. The diagonal matrix Σ has entries called the singular values σ_i of A. The singular values $\sigma_1 \geqslant \cdots \geqslant \sigma_n \geqslant 0$ has corresponding right and left singular vectors $u_i \in U$ and $v_i \in V$ of A. See [24, 41] for details and proof.

Every matrix admits an SVD. However, if A is a rank $r < n$ matrix, the remaining $(n-r)$ singular values will be zero. Hence, the remaining $(m-r)$ columns of U and $(n-r)$ columns of V are superfluous, as they will be multiplied by zero. Therefore, if A is not full rank, the SVD is not unique. To see this, choose $\tilde{U} = UP$ and $\tilde{V} = VQ$ where both P and Q are orthogonal we loose uniqueness if $\tilde{S} = P^TSQ$. This yields $Y = USV^T = \tilde{U}\tilde{S}\tilde{V}^T$. As a consequence, the matrices U, V and Σ can be truncated without loss of information. We call this the truncated or *thin* SVD [24].

In fact, the leading singular values and vectors contain the most "information" so that truncating the matrix can be done with minimal loss. Say we have a matrix $A \in \mathbb{R}^{m\times n}$ of rank r, and we want to find a low-rank $k < r$ approximation A_k of A, that is

$$\min_{A_k\in\mathcal{M}_k} ||A_k - A||_F, \tag{2.3}$$

where $\mathcal{M}_k^{m\times n}$ denotes the set of all $m \times n$ matrices of rank k. In fact the truncated SVD is the solution to (2.3); the best approximation A_k to A.

That implies that A_k is given by

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T. \tag{2.4}$$

This is known as the Eckart-Young-Mirsky theorem [18]. Another important result from this theorem is that the error in the low-rank approximation is given by the largest omitted singular value;

$$||A - A_k||_F = \sigma_{k+1}.$$

Moreover, the result also holds for matrix-valued functions $A(t), t \in [0, T]$ [52]. The solutions U and V of (2.3), (2.4) are optimisation problems on the *Stiefel* manifold, where the constraint is orthonormal matrices embedded in $\mathbb{R}^{m\times k}$ for U and $\mathbb{R}^{n\times k}$ for V [2]. We define the Stiefel manifold as follows

$$St(n, p) = \{U \in \mathbb{R}^{n\times p} : U^T U = I_p\}.$$

Given the Singular Value Decomposition (2.2), we have by inserting V^*V

$$A = U\Sigma V^* = U \underbrace{(V^*V)}_{=I} \Sigma V^* = \underbrace{W}_{UV^*} \underbrace{P}_{V\Sigma V^*},$$

the *Polar Decomposition*. As W is the product of two unitary matrices, W is also unitary; $W \in St(m, n)$. Furthermore, as Σ is positive semi-definite, P is as well. Given a matrix $A \in \mathbb{R}^{m\times n}$, the first factor from the polar decomposition is the best orthogonal approximation to A [40]. This polar factor W can be found either from the SVD or from iteration methods proposed in [38]. We will get back to the polar transformation in the next chapter.

The SVD is a linear transformation, and as we cannot assume the manifold to be Euclidean, we need tools for non-linear dimension reduction. In this thesis, one of the main methods we will use is the SVD combined with a Residual Neural Network, presented in the next chapter. A similar approach can be seen in a paper from 1993 [49]. The authors combined a five-layer auto-associative network presented by Oja in 1991 [66] with PCA to produce a local linear method for non-linear dimension reduction. PCA alone typically fails, as seen in [69]. The author argues that PCA does not reflect the typical variation in a data set. One can suspect that SVD will have similar flaws as a method on its own.

We seek to reduce the dimension of the images A. If we consider the truncated decomposition of A such that we have thin and wide $U_k \in St(m,k), V_k \in St(n,k)$ and $\Sigma \in \mathbb{R}^{k\times k}$ non-singular, we have achieved a dimension reduction. But in order to utilise this dimension reduction we are dependent on our model to be able to handle the separate matrices. Lastly, this is the minimal loss approach to the original matrix while maintaining the basis for the subspace and other useful properties of A. A continuous extension to the SVD will be presented in the following subsection.

2.3.2 Dynamic Low-rank Approximation

Instead of finding a low-rank approximation of the data $A \in \mathbb{R}^{m\times n}$, see Equation (2.3), we can derive a low-rank approximation for the derivative $\dot{A}(t)$. This method was proposed by Koch & Lubich in 2005 [52]. The method exploits the fact that $A(t)$ is time dependent. The approximation is obtained using a factorised form solving the differential equation for the three factors, U, Σ and V.

Assume there exists a differentiable matrix-valued image classification function $A(t) \in \mathbb{R}^{m\times n}$ with respect to t, which is able to transform and classify the images correctly. The goal is to find a low-rank approximation $Y(t) \in \mathcal{M}_k^{m\times n}$ to this function. We reformulate the problem as

$$\min_{\dot{Y}(t)\in\mathcal{T}_{Y(t)}\mathcal{M}_k} ||\dot{Y}(t) - \dot{A}(t)||_F. \tag{2.5}$$

The low-rank approximation $Y(t) \in \mathcal{M}_k$ can be determined from its derivative $\dot{Y}(t) \in \mathcal{T}_{Y(t)}\mathcal{M}_k$ by integration with this approach. The low-rank differential equation has a solution evolving on the low-rank manifold. One can think of this as the *continuous* low-rank approximation of the classification function $A(t)$.

By a differential equation evolving on $\mathcal{M}$ we mean a differential equation of the form

$$\dot{y} = F(t,y), \quad t \geqslant 0, \quad y(0) \in \mathcal{M}, \tag{2.6}$$

where $F \in \mathcal{X}(\mathcal{M})$ is the set of all vector fields on the manifold. The *flow* of F is the operator $\Psi_{t,F} : \mathcal{M} \to \mathcal{M}$ such that

$$y(t) = \Psi_{t,F}(y_0) \tag{2.7}$$

where $y(t)$ is the solution of(2.6). The flow is obtained by computing the *exponentiation* of the vector field. Computation of the flow is a particular example of an *exponential map* [46].

Therefore, (2.5) can be can be seen as the orthogonal projection of $\dot{A}(t)$ on $\mathcal{T}_{Y(t)}\mathcal{M}_k$ with respect to the Frobenius inner product. This orthogonality condition can be expressed as

$$\langle \dot{Y} - \dot{A}, \delta Y \rangle = 0, \quad \forall \delta Y \in \mathcal{T}_Y \mathcal{M}_k. \tag{2.8}$$

where the time dependence of $\dot{Y}(t)$ and $\dot{A}(t)$ has been omitted for ease of notation. To find Y(t) we need to solve an initial value problem of non-linear ODEs on $\mathcal{M}_k$, complemented with the initial condition $Y(t_0) = A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$. We cannot expect $Y(t)$ to remain close to $A_k(t)$, which is obtained by performing the SVD at every time step [52], and this is also not necessarily the goal. We will later compare the two methods, the dynamic low-rank approximation solution $Y(t)$ to SVD at every time step $A_k(t)$.

We seek a unique and continuous low-rank approximation factorised on the form

$$Y(t) = U(t)S(t)V(t)^T, \tag{2.9}$$

where $U(t) \in St_{m,k}$ and $V(t) \in St_{n,k}$ with orthonormal columns evolve on the Stiefel manifold [29]. Lastly, $S \in \mathbb{R}^{k\times k}$ as seen in (2.2) is a diagonal matrix, but here it is only assumed to be non-singular. To circumvent the issue of non-uniqueness in equation (2.9), as seen in the SVD, we choose a unique decomposition in the tangent space. Taking the derivative of the matrix $Y = USV^T$ we find an expression for the tangent space $\delta Y \in \mathcal{T}_{\mathcal{Y}}\mathcal{M}_k$

$$\delta Y = \delta U S V^T + U \delta S V^T + U S \delta V^T, \tag{2.10}$$

where $\delta S \in \mathbb{R}^{k\times k}, \delta U \in \mathcal{T}_U \mathcal{V}_{m,r}$ and $\delta V \in \mathcal{T}_V \mathcal{V}_{n,r}$. Lastly, by imposing the orthogonality constraints $U^T \delta U = 0$ and $V^T \delta V = 0$, the matrices δS, δU and δV are now uniquely determined by δY. Rewriting the equations and using (2.10) together with the orthogonality constraints we obtain the corresponding equations in Theorem (2.1). The argument is described in detail in [52].

Theorem 2.1 (Dynamic low-rank Approximation)
For $Y = USV^T \in \mathcal{M}_k$ with nonsingular $S \in \mathbb{R}^{k\times k}$ and with $U \in \mathbb{R}^{m\times k}$ and $V \in \mathbb{R}^{n\times k}$ having orthonormal columns, condition (2.5) or (2.8) is equivalent to $\dot{Y} = \dot{U}SV^T + U\dot{S}V^T + US\dot{V}^T$, where

$$\begin{aligned} \dot{S} &= U^T \dot{A} V \\ \dot{U} &= (I - UU^T)\dot{A}VS^{-1} \\ \dot{V} &= (I - VV^T)\dot{A}^T US^{-1} \end{aligned} \tag{2.11}$$

Proof. [52]

Theorem (2.1) and the method it proposes is related to other matrix-valued differential equations, in particular the smooth SVD presented in [14, 47].

A natural extension to (2.5) is the so-called minimum-defect low-rank approximation [29]. The matrix-valued differential equation $\dot{A} = F(A)$ can be approximated by replacing $\dot{A}$ in (2.5) by the approximation $F(Y(t))$,

$$\min_{\dot{Y}(t) \in \mathcal{T}_Y \mathcal{M}_k} ||\dot{Y} - F(Y)||_F. \tag{2.12}$$

The approach of (2.12) is what we will continue with in the next chapter. There we will use the low-rank approximation to approximate a vector field generated and trained by a neural network. There we will also present a first order integration method for the system (2.11).

2.3.3 Dynamical Tensor Approximation

Until now, we have only considered matrices. If we desire to evaluate images of colour, additional dimensions need to be added. Most commonly, images in RGB (Red/Green/Blue) have additional dimensions, such that for each *colour channel* there is a matrix of the image size. This leaves us in a position where an extension to tensors is needed. An extension of the low-rank approximation to tensors can be found in [53, 64]. The first reference, whose title is shared with this subsection, is the one we will discuss.

Similarly to the Dynamic Low-Rank approach (2.5) and the minimum-defect low-rank approximation, (2.12) one can ask if there exists a low-rank approximation for tensors. For a fixed tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, there are many possible decompositions and approximations [54]. One of these is the Higher-Order Singular Value Decomposition (HOSVD), commonly referred to as the Tucker Decomposition, as it was introduced by Tucker in 1963 [78]. The Tucker decomposition is a low-rank approximation. However, determining if this is the *best* low-rank approximation is not an easy task, we are not even sure if it exists [54]. The tensor equivalent to matrix rank is NP-hard to compute, as shown by Håstad in 1990 [33]. Therefore, there does not exist an efficient algorithm to determine the rank of a tensor. Before we define the Tucker Decomposition, we need to set some notation and vocabulary. We will follow the definitions and notations from [54, 53].

Let $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. The number of dimensions of a tensor is referred to as the *order* or *modes*. The tensor equivalent to rows and columns are *fibers*; see Figure 2.2 in [54] for good illustrations. The norm of a tensor is analogous

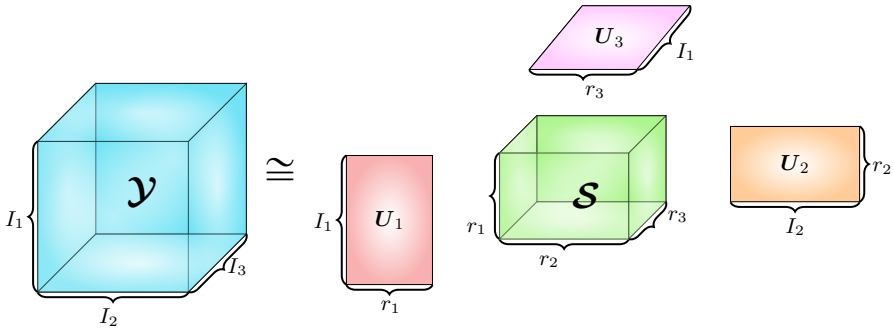


Figure 2.2: Illustration of the Tucker Decomposition of a tensor.

to the Frobenius norm of a matrix. The nth *unfolding* (matricization) of $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is the process of reordering the elements of the tensor into a matrix. The nth unfolding is denoted $\boldsymbol{Y}_{(n)} \in \mathbb{R}^{I_n \times I_{n+1} \ldots I_N I_1 \ldots I_{n-1}}$, where we arrange the mode-n fibers as columns in the resulting matrix. Note that for each mode of the tensor, there exists an unfolding. The *n-mode (matrix) product* is the multiplication between a tensor and a matrix in mode n. Let $B \in \mathbb{R}^{J \times I_n}$ be a matrix. The n-mode matrix product of the tensor $\boldsymbol{\mathcal{Y}}$ and the matrix B is denoted by $\boldsymbol{\mathcal{Y}} \times_n B$. Each mode-$n$ fiber is multiplied by the matrix B and the result is of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$. This implies that the nth mode of the tensor must match the second mode of the matrix. We are now ready to introduce the Tucker Decomposition. The Tucker Decomposition decomposes the tensor $\boldsymbol{\mathcal{Y}}$ into a core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{r_1 \times \cdots \times r_N}$ and N-matrices, U_i for $i = 1, \ldots, N$, along each mode, see Figure 2.2. Every tensor $\boldsymbol{\mathcal{Y}}$ admits a Tucker Decomposition [11]. The Tucker decomposition is defined as

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{S}} \times_1 U_1 \times_2 \cdots \times_N U_N := \boldsymbol{\mathcal{S}} \bigtimes_{n=1}^{N} U_n,$$

$U_n \in \mathbb{R}^{I_n \times r_n}$, and are orthogonal if the algorithm used to compute the decomposition is the SVD. The dimensions $r = \{r_1 \times \ldots r_N\}$ is known as the *Tucker rank* of the tensor. Just as we can choose a truncation k in the SVD we choose the Tucker rank r in the Tucker Decomposition. To show the relation between the Tucker Decomposition and the SVD, we can write it in terms of the nth unfolding

$$\begin{aligned} \boldsymbol{Y}_{(n)} &= \boldsymbol{U}_{(n)} \boldsymbol{S}_{(n)} (\boldsymbol{U}_{(N)} \otimes \cdots \otimes \boldsymbol{U}_{(n+1)} \boldsymbol{U}_{(n-1)} \otimes \cdots \otimes \boldsymbol{U}_{(1)})^T, \\ &:= \boldsymbol{U}_{(n)} \boldsymbol{S}_{(n)} \bigotimes_{k \neq n} \boldsymbol{U}_{(k)}^T, \end{aligned}$$

where $\otimes$ denotes the Kronecker matrix product. Similarly to the SVD, the Tucker decomposition is not unique [53].

Consider a time-dependent tensor $\boldsymbol{\mathcal{A}}(t) \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, for $0 \leqslant t \leqslant T$. Let $\boldsymbol{\mathcal{M}}_r$ be the manifold of Tucker rank r matrices

$$\boldsymbol{\mathcal{M}}_r = \{\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_N} : \boldsymbol{\mathcal{Y}} \quad \text{has Tucker rank } \boldsymbol{r}\}.$$

We can finally state the minimisation problem, known as the *dynamical tensor approximation*, where $\boldsymbol{\mathcal{Y}}(t) \in \boldsymbol{\mathcal{M}}_r$ is determined from (2.13)

$$\min_{\dot{\boldsymbol{\mathcal{Y}}}(t) \in \mathcal{T}_{\boldsymbol{\mathcal{Y}}(t)} \boldsymbol{\mathcal{M}}_r} ||\dot{\boldsymbol{\mathcal{Y}}}(t) - \dot{\boldsymbol{\mathcal{A}}}(t)||, \tag{2.13}$$

by linear projection. Similarly as in the matrix case, we end up with a system of $N+1$ non-linear ordinary differential equations on $\boldsymbol{\mathcal{M}}_r$, with the initial condition $\boldsymbol{\mathcal{Y}}(0) = \boldsymbol{\mathcal{X}}(0)$ in low-rank Tucker format. The arguments from the matrix case can be extended to tensors, as seen in [53]. We will state the result here

Theorem 2.2 *For a tensor $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{S}} \times_{n=1}^{N} U_n \in \boldsymbol{\mathcal{M}}_r$ with rank $r = \{r_1, \ldots, r_N\}$, core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{r_1 \times \cdots \times r_N}$ and n-mode factors $U_n \in \mathbb{R}^{I_n \times r_n}$ having orthonormal columns, condition (2.13) is equivalent to*

$$\dot{\boldsymbol{\mathcal{Y}}} = \dot{\boldsymbol{\mathcal{S}}} \bigtimes_{n=1}^{N} U_n + \sum_{n=1}^{N} \boldsymbol{\mathcal{S}} \times_n \dot{U}_n \bigtimes_{\boldsymbol{n \neq k}} U_k,$$

where the factors in the decomposition satisfy the system of differential equations

$$\dot{\boldsymbol{\mathcal{S}}} = \dot{\boldsymbol{\mathcal{A}}} \bigtimes_{\boldsymbol{n=1}}^{\boldsymbol{N}} U_n^T, \tag{2.14}$$

$$\dot{U}_n = (I - U_n U_n^T)[\dot{\boldsymbol{\mathcal{A}}} \bigtimes_{\boldsymbol{n \neq k}} U_k^T]_{(n)} S_{(n)}^{\dagger}, \tag{2.15}$$

where $S_{(n)}^{\dagger} = S_{(n)}^T (S_{(n)} S_{(n)}^T)^{-1}$ is the pseudo-inverse of the n-mode unfolding $S_{(n)}$ of $\boldsymbol{\mathcal{S}}$.

Proof. [53]

2.4 Closing remarks

Throughout this chapter, we have motivated the manifold hypothesis and methods which take advantage of it. Along with the methods mentioned in this chapter and our neural network, there exist many methods which deal with manifold learning and reduced-order models. In [9], the author suggests that the primary reason for the many algorithms is due to difficulties in evaluating their performance. In most cases, in order to evaluate a specific algorithm, it is run on an artificial data set and see if the result is "intuitively pleasing". The theoretical estimates are typically coarse and few. Lastly, if we are able to study the data set on the lower-dimensional manifold we hope that we gain some robustness to *adversarial attacks*.

Chapter 3

Deep Learning as Optimal Control

In the twenty-first century, the robot will take the place which slave labor occupied in ancient civilization.
– Nikola Tesla

The goal of this chapter is to introduce Neural Networks implementing the methods from the previous chapter. First, we introduce and motivate neural networks and the classification problem. Then we will draw the connection between the Residual Neural Networks and the discretised ODE and formulate the continuous learning problem. From the continuous learning problem we will present methods for geometric numerical integration of the methods from the previous chapter. Three networks for matrices and tensors will be presented. These are the main methods of this thesis. Lastly, we will discuss adversarial attacks as a way to measure robustness of these methods.

3.1 Classification using Neural Networks

When we see a picture of a blurry cat or a number in funny handwriting, we are often immediately able to tell that it is, in fact, a cat and what number it is. This task seems easy for a human to solve, but the question has been for decades if we can enable computers to . In Figure 3.1, the blurry number 8 is shown in a 28×28 grid, where the numbered cells indicate colour. Can we find a function, or a computer program, that will take grids as input and output a number? In machine learning, the goal is to learn functions

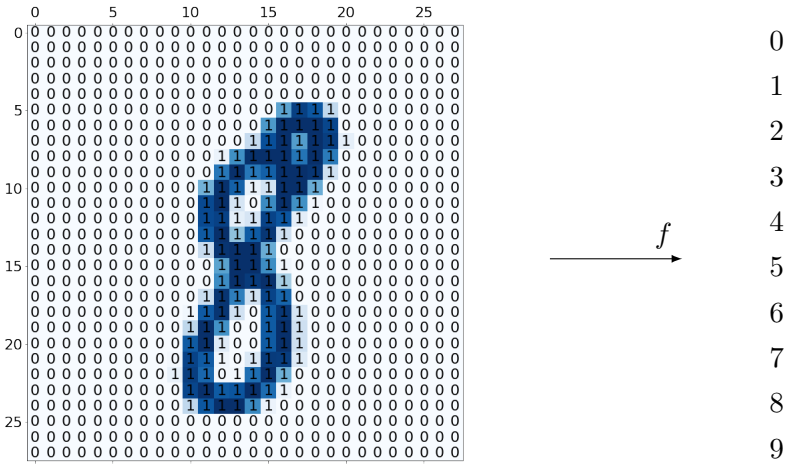


Figure 3.1: Map from an image of 28×28 pixels to a real number.

known as *classifiers*; i.e a function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$, such that given inputs are mapped onto a set of labels [6]. One could call this a function on the grid determined by the positions (i, j) for $i, j = 1, \ldots, 28$ in the matrix, corresponding to certain points (x_i, y_j) in $\mathbb{R}^{m \times n}$. In this example, we use the interpolated model found to generalise and extrapolate to unseen data.

3.1.1 Neural Networks

Neural networks constitute a broad and rich class of models that have become standard in image classification, speech recognition and many other applications [59]. The name, neural network, is inspired by the brain where groups of neurons are connected. The definition presented by Fiesler in [20] is broad enough to describe all artificial neural networks as well as biological ones. When we discuss neural networks, we will imply artificial neural networks only. We will now present an overview of neural networks.

A *feed-forward* neural network, also known as a *multilayer perceptron (MLP)*, is a collection of nodes and edges between them. We assume the edges do not form cycles of nodes. Information travels only one way; forward [44]. A simple neural network is shown in Figure 3.2. We cluster the nodes, often called neurons, in L layers, where layer l is a collection of nodes $d^{[l]}$ [20]. In Figure 3.2 the network is *fully connected*. With this structure every neuron in the previous layer is connected to every neuron in the present layer; see also Figure 3.3.

Let $\mathcal{X}^{[l]}$ denote a vector space for any layer $l \in \{0, 1, \ldots, L-1\}$ [10]. The

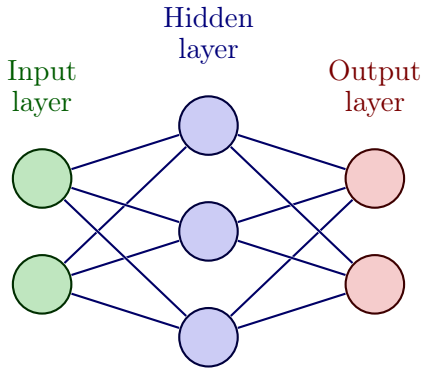


Figure 3.2: Simple Neural Network.

input layer $\mathcal{X}^{[0]} = \mathcal{X}$, in green, takes in the input data. If the input is a matrix in $\mathbb{R}^{m \times n}$ we require a minimum $m \times n$ input nodes. We denote the input data $\{\boldsymbol{x}_i\}_{i=1}^{N}$ and the corresponding nodes as $x_j \in \boldsymbol{x}_i$. The *hidden layers* $\mathcal{X}^{[l]}$ for $0 < l < L$, are the layers in blue. We say we have hidden layers if $L \geqslant 2$, then the network is referred to as deep [37]. In Figure 3.2, we only have one hidden layer. We can choose the number of hidden layers $L-1$, called the *depth* of the neural network. One can also choose the number of neurons in each layer $d^{[l]}$, called the *width* of the layer. Lastly, we have the *output layer* $\mathcal{X}^{[L]} = \mathcal{Y}$ in red. The nodes in the output layer represent labels. We have one node for each class $k \in K$. We denote the output as $\boldsymbol{y}_i \in \mathbb{R}^K$. The *transition function* f describes how to propagate through the network and combine the layers from the initial state. One could also view the network as a combination of simple parametric functions between layers or *feature spaces*. We can define the neural network as the iteration between parameterised layers as follows

$$\begin{aligned} \Psi : \mathcal{X} \times \Theta &\to \mathcal{Y}, \\ (x, \theta) &\mapsto y^L, \\ f^{[l]} : \mathcal{X}^{[l]} \times \Theta^{[l]} &\to \mathcal{X}^{[l+1]}, \end{aligned}$$

where $\Theta^{[l]}$ is the set of possible parameter values for layer l. This combination is typically done using function composition [10]. Lastly, if the number of classes is smaller than the dimension of the input, $K < m \times n$, we need to include a layer $\eta : \mathcal{X} \to \mathcal{Y}$ [6]. The neural network can be interpreted as a composition of layers $y^{[L]} = f^{[L-1]}(y^{[L-1]}, \theta) \circ \cdots \circ f^{[0]}(y^{[0]}, \theta)$ where $y^{[0]} = x$; simple functions that can represent complicated ones. And now

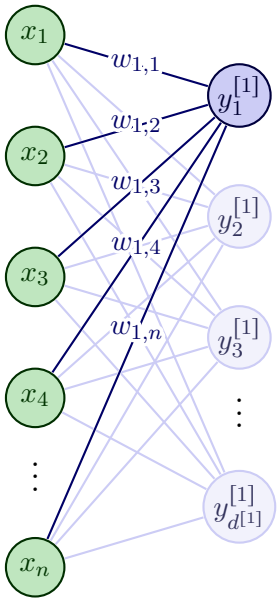
the we can write the neural network as an iteration of the form

$$y^{[0]} = x, \tag{3.1}$$
$$y^{[l+1]} = f^{[l]}(y^{[l]}, \theta^{[l]}). \tag{3.2}$$

We have some immediate restrictions on a neural network. We do not want an infinite amount of neurons, and the neurons typically hold values between zero and one; $x_i \in [0, 1]$. Hence, we have finite-dimensional Euclidean vector spaces as layers in the network. The value of the neuron i at layer j is called the *activation* $y_i^{[j]}$. In the case of Figure 3.1, the pixel value corresponds to the activation in the input layer. For the output, the highest activation can be thought of as the network's "choice" of label or the probability that the label is correct. The activation of all the nodes in the layer $\mathcal{X}^{[l-1]}$ determine the activations in layer $\mathcal{X}^{[l]}$. This is true for all layers, see Figure 3.3. To determine the activation of a neuron in this layer, we multiply the activation from the nodes in the previous layer with the *weights* of the connections between them. The input layer, containing n nodes with the activation $\boldsymbol{x}_i = \{x_1, x_2, \ldots, x_n\}$, is connected to the first hidden layer by a set of weights collected in the matrix $W^{[0]} = [\boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_n}] \in \mathbb{R}^{n \times d^{[1]}}$. The number of elements in each vector $\boldsymbol{w_i} \in \mathbb{R}^{d^{[1]}}$ is determined by the number of neurons in the first hidden layer $d^{[1]}$; see Figure 3.3 [21]. The neuron's activation measures of how positive the weighted sum is. Lastly, we add a *bias* $\boldsymbol{b}^{[l]} = \{b_1, \ldots, b_{d^{[l]}}\} \in \mathbb{R}^{d^{[l]}}$ to the weighted sum. If the value passes a threshold, it will be activated. Writing all the equations for the neurons activation's in matrix-vector from we arrive at the equation seen in Figure 3.3. The activations in each layer can be expressed as

$$\begin{aligned} y^{[1]} &= \sigma(W^{[0]} x^{+} b^{[0]}), \\ y^{[2]} &= \sigma(W^{[1]} y^{[1]} + b^{[1]}), \\ &\vdots \\ y^{[L]} &= \sigma(W^{[L-1]} y^{[L-1]} + b^{[L-1]}), \end{aligned}$$

where the activation function σ acts component-wise. Note, however, that we do not typically put a bias vector on the input layer, leaving $\boldsymbol{b}^{[0]} = \boldsymbol{0}$. The *activation function* $\sigma : \mathbb{R} \mapsto \mathbb{R}$ maps the linear combination of the activation from the previous onto the interval $[0, 1]$ for each neuron. The activation function σ is a non-linear transformation. With this added non-linearity, we enable the network to work around the fact that linear models can only separate the input space into simple regions easily separated by a



$$\begin{pmatrix} y_1^{[1]} \\ y_2^{[1]} \\ \vdots \\ y_d^{[1]} \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d,1} & w_{d,2} & \dots & w_{d,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} b_1^{[0]} \\ b_2^{[0]} \\ \vdots \\ b_n^{[0]} \end{pmatrix} \right],$$

$$y^{[1]} = \sigma\left(\mathbf{W}^{[0]}x + \mathbf{b}^{[0]}\right).$$

Figure 3.3: *Left:* Naming convention for weights and and activations from one layer to another, here input- to first layer. *Right:* Graph to the left translated into the full equation in matrix-vector form.

hyperplane [44]. This is crucial, as more complex problems such as image and speech recognition require the network to be insensitive to irrelevant input variations, such as noise. But sensitive to differences between a toy cat and a real cat. This is the selectivity-invariance dilemma; we require a selective model for patterns that are important for discrimination and invariant to irrelevant details [44].

Equation (3.3) shows examples of activation functions to choose from;

$$\begin{aligned} \text{logit}(x) &= \frac{1}{1+\exp -x}, & \text{ReLu}(x) &:= max(0,x), \\ \text{softplus}(x) &:= \log(1+\exp x), & \tanh(x) &:= 2\text{logit}(2x)-1. \end{aligned} \tag{3.3}$$

Experiments have shown that the choice of activation function does not have much of an impact on neural networks' performance [21]. A paper from 2011 [23] on the ReLU function showed that very deep neural networks benefit in computational speed if the ReLU function is used in supervised training.

Now, the network is taking form as a stack of pretty simple modules. Each layer contains *trainable parameters* and non-linear maps that increase the input's selectivity and invariance. It is clear that with many layers, the rep-

resentation power is increased. This enables the network to learn intricate functions that classify complex problems. This is the key advantage of *deep learning* [59]. When the depth and width of the network increase, the number of trainable parameters also increases. In [73], the author showed that standard network types always gain representation power when the number of layers increase. However, this comes at the cost of enormous computation power. Therefore, we seek smaller networks with fewer trainable parameters with equal performance. A natural question to ask is how to determine the optimal width and depth of the network. These questions will not be addressed in this thesis.

We will now leave behind the discussion of individual nodes in the network, and use this notation to describe individual data points and images. With input data $x_i = x \in \mathbb{R}^n$, where $n \in \mathbb{N}$. Now we have, $d^{[0]} = n$, and for all layers $l = \{0, \dots, L-1\}$ where the weight matrix $W^{[l]} \in \mathbb{R}^{d^{[l+1]} \times d^{[l]}}$ and bias $b^{[l]} \in \mathbb{R}^{d^{[l+1]}}$ [21]. The *forward propagation* of the data through the network is defined as (3.2), with linear layers $f^{[l]}(y^{[l]}, \theta^{[l]}) := \sigma(W^{[l]}y^{[l]} + b^{[l]})$. Consider the classification problem with K classes; $y_i = y^{[L]} \in \mathbb{R}^K$. Now it is clear that Equation (3.2) leads to an algorithm for feeding input through the network. The algorithm produces output in $y^{[L]} \in \mathbb{R}^K$, when combined with the layer η. Now we only lack the training procedure [37]. When we say we are training the network we are essentially solving the optimisation problem of finding the optimal weights and biases of the network for correct classification.

3.1.2 Training the neural network:

Assume a *training data* set $N := (x_i, c_i)_{i=1}^N$ of size $N \in \mathbb{N}$ consisting of samples $x_i \in \mathbb{R}^d$ and labels $c_i \in \mathbb{R}^K$, where $\boldsymbol{c}_i$ has the dimension of the output layer and is thus comparable to the network's output $y_i \in \mathbb{R}^K$ [21]. This data set contains examples x_i of a problem we want the network to be good at solving, and also the true label c_i such that the network can verify if it predicted right or not. If the network did not predict correctly, we will have to adjust the weights and biases such that next time it will give a correct prediction given this exact input. If it did predict correctly, it will strengthen the weights and biases such that given this input next time, it will be even more confident of its guess. After the network has achieved optimal parameters based on the training data, we feed the network with *validation data*, $V := (x_i, c_i)_{i=1}^V$ of size $V \in \mathbb{N}$, and test how well it performs on data it has never encountered before. We measure the accuracy, $\frac{correct}{total}$, for both the training set and the test set to get an idea of how good the

overall performance is.

We start by initialising the parameters. The initial trainable parameters are set to zero or sampled from a distribution. The choice of distribution will impact the network's training, as shown in [30]. Now, given input data, the network will give a prediction in the output layer, but most likely, it will be incorrect. The error can be measured using the residual sum of squares (RSS) (3.5). We call this *the cost function* (3.4) where $J(\Theta) : \mathbb{R}^P \rightarrow \mathbb{R}$, where P is the number of trainable parameters in the network. We want to minimise the error it makes and thus find the minimum of the cost function as we adjust the weights and feed it training data. We formulate the minimisation of the cost function as follows;

$$\min_{W,b} \Big\{ J(W,b) + \mathcal{R}(W,b) \Big\}, \tag{3.4}$$

$$\textit{RSS} \qquad J(W,b) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} ||c_i - y^{[L]}||_2^2, \tag{3.5}$$

$$\textit{Cross-Entropy} \quad J(W,b) = -\frac{1}{N} \sum_{i=1}^{N} \big(c_i \cdot \log(y^{[L]}) \big). \tag{3.6}$$

The cross-entropy loss, Equation (3.6), is frequently used over RSS as it penalises really bad estimates. We can also add a term $\mathcal{R}(W,b)$ as seen in (3.4), called a regularisation function that will ensure more control and regularity of the parameters W, b [6]. Two options for the regularisation function are techniques known from statistics as Ridge regression and Lasso [76], known in machine learning as weight decay [44]. The Lasso (Least Absolute Shrinkage and Selection Operator) technique uses the L^1 penalty with a linear model. Regularisation is also a solution if our problem at hand is prone to *overfitting* [37]. However, regularisation comes with a price, as even for small terms $\mathcal{R}$, the performance of the neural net deteriorates, especially for the validation data.

The efficient computation of (3.4) is called *backpropagation*, and the method is often stochastic gradient descent or the ADAM method [51]. We find the derivatives with respect to the trainable parameters and minimise. We divide and shuffle our data into *mini-batches* and then optimise the network for each batch. We do this to minimise the network learning a sequential pattern, and each mini-batch gives a good approximation and improves computational speed. If the number of trainable parameters $W^{[l]}, b^{[l]} \forall l$ and the data set

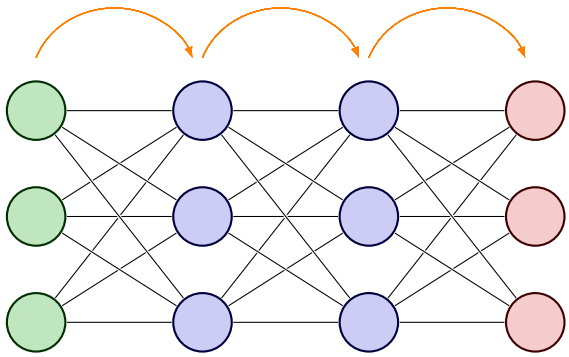


Figure 3.4: Simple Residual Neural Network with two hidden layers of constant width. The orange arrow indicates the residual connection.

N is large, the task of computing the gradient vector is costly. Hence, we end up minimising a single random gradient representing the complete data set. This is known as vanilla stochastic gradient descent, and there are many ways to optimise and vary the method. Now, we have both network architecture and a way to propagate, optimise, and train the network by updating the parameters.

3.2 Residual Neural Networks and Neural ODEs

A new deep learning architecture was proposed by He et.al in 2016 [35], called Residual Neural Networks (ResNet). These networks were introduced to improve learning in very deep networks, for instance 100 layers [35]. They introduced a residual connection, an *identity map*, between layers. It performs well when the number of layers is high, and performs better than traditional feed-forward networks, defined previously (3.2). These deep networks long performed much worse than shallower models. He et al. achieved state-of-the-art results using ResNets with 152 convolutional layers on image recognition. Arguably, the increased performance stems from the presence of the identity mapping that gives improved performance in the backpropagation [36].

The ResNet architecture is made from *stacked units*; given initial data $x = y^{[0]} \in \mathcal{X}^0$, the Residual Neural Network $\Psi : \mathcal{X} \times \Theta \to \mathcal{Y}$ where $\Psi(x, \theta) = y^{[L]} \in \mathcal{Y}$ is defined by the iteration $l = \{0, 1, \dots, L-1\}$

$$\begin{aligned} y^{[0]} &= x, \\ y^{[l+1]} &= y^{[l]} + \sigma(W^{[l]} y^{[l]} + b^{[l]}). \end{aligned} \tag{3.7}$$

Comparing this new architecture (3.7) to the one we discussed earlier (3.2), we immediately notice that the only change is the addition of the previous layer $y^{[l]}$ to the new layer $y^{[l+1]}$; the identity map, see Figure 3.4. Note that the number of neurons have to be equal in all ResNet layers.

The ResNet formulation can be viewed as the discretisation of a time- and parameter dependent differential equation. To justify the relation of Equation (3.7) to ODEs, let y, W and b be the time discretisation of the continuous variables $y(t) : [0,T] \to \mathbb{R}^d, W(t) : [0,T] \to \mathbb{R}^{d\times d}, b(t) : [0,T] \to \mathbb{R}^d$. Define $f(y(t),\theta(t)) := \sigma(W(t)y(t) + b(t))$, such that

$$y^{[l+1]} = y^{[l]} + hf(y^{[l]},\theta^{[l]}), \tag{3.8}$$

where $h = 1$ if $L = T$ and $l = \{0,\dots,L-1\}$ is a partition of the interval $[0,T]$. Let $h := \Delta t$ and $y^{[0]} := y(0)$. Rewriting the equation (3.8), gives us

$$\frac{y^{[l+1]} - y^{[l]}}{\Delta t} = f(y^{[l]},\theta^{[l]}).$$

Let $\Delta t \longrightarrow 0$, and we arrive at the *continuous* forward propagation

$$\frac{dy(t)}{dt} = f(y(t),\theta(t)) = \sigma(W(t)^T y(t) + b(t)), \quad \forall t \in [0,T] \tag{3.9}$$

where T is the time corresponding with the output layer L. The model parameters $\theta = (W,b)$ are now required to match the introduced functions at time $t_j = jh$, such that $y(t_j) = y^{[l]}$ and $\theta(t_j) = \theta^{[j]}$ for layer and time step j [6, 21]. We can analyse the continuous formulation of (3.9) in the general ODE framework. The ResNet formulation (3.7) is now recognised as the forward Euler discretisation of the general ODE

$$\begin{aligned} y(0) &= x \\ \dot{y}(t) &= f(y(t),\theta(t)), \quad t \in [0,T]. \end{aligned}$$

As a consequence, we can establish a connection between the ResNet (3.7) and an *optimal control* problem. In optimal control, the goal is to optimise a functional of a dynamical system, subject to an ODE constraint. A dynamical system is defined by a function $f : \mathbb{R}^d \to \mathbb{R}^d$. We define the optimal control problem as

$$\min_{u\in\mathcal{U}} \mathcal{J}(y,u) \quad \text{s.t} \quad \dot{y}(t) = f(y(t),u(t)), \quad y(0) = y_0, \tag{3.10}$$

where $y(t)$ is called the state of the system at time t. The control $u(t)$ is a time- varying function, independent of y, altering the behaviour of f. The

goal is to minimise the cost functional $\mathcal{J}(y,u)$ by determining $u \in \mathcal{U}$ in a set of admissible controls [71]. For existence and uniqueness of optimal control problems on the form (3.10), see the book by Sontag [71]. We will assume existence of solutions to the optimal control problem, and focus solely on the numerical aspect of the problem.

We assume that the cost function is only dependent of the last time-state. We summarise the deep learning optimal control problem

$$\min_{y,u,\hat{W},\hat{b}} \sum_{i=1}^{N} |\mathcal{C}(\hat{W} y_i^{[L]} + \hat{b}) - c_i|^2, \tag{3.11}$$

subject to the constraint

$$\dot{y}_i = f(y_i, u_i), t \in [0,T], \quad \begin{cases} y_i(0) = x_i, \\ y_i(T) = y_i^{[L]} \end{cases} \tag{3.12}$$

where $u = (W,b)$, and $\mathcal{C}$ is the *hypothesis* function. The parameters $\hat{W} : \mathbb{R}^d \to \mathbb{K}$, $\hat{b} : \mathbb{R}^d \to \mathbb{K}$ reduce the output from last layer into a size comparable to c_i. The hypothesis function, typically the softmax (normalised exponentional function), maps the output of the network to a probability vector that can be compared to the class label c_i [6].

From a numerical point of view, we can see the neural network as an approximation method or a function approximator. In numerical approximation schemes; the error will decrease if we decrease the step size. From The neural net iteration, Equation (3.2). We will approach the deep learning optimal control problem from the view of numerical analysis. We have the continuous optimal control problem, for details see (3.11), (3.12), and need to discretise it to get our algorithm for the training problem. We can either *optimise then discretise*, or *discretise then optimise* [6]. Typically, we first dicretise then optimise, and this is the approach we will take in the next subsection. Following this approach gives automatically the formulae for back-propagation, and the gradient of the cost function as we discussed previously, as seen in [6, 21].This allows us to use automatic differentiation. Using the first optimise then discretise approach gives us explicit formulas for the gradient, but this will not be considered here.

Before we move into the numerical procedure we include a remark on the connection between topology and ResNets. In the ResNet framework we are training a vector field to move data points such that the network can classify them. The network recognises differences, and separates them in

the feature space. In the manifold setting presented in the previous chapter, our neural network (vector field) is moulding our manifold in such a way that it is capable of classifying parts of it. In topology, we use continuous deformation as a tool to classify different objects. The topology of the data may impose challenges to this simple idea and a paper presented by Dupont [16] illustrates the limitations such continuous deformations have. The trajectories of the vector field cannot cross, and therefore it will in some cases have problems with classifying nested structures. An example is linked tori, or nested circles as seen in [16], the authors prove that there are functions the Neural ODEs cannot represent. Therefore increasing the dimension of the feature space, and thus the vector field, will allow it to move things from inside holes to the outside of the holes. How can we determine the number of nested structures in our image manifold? Can this enable us to determine the number of additional dimensions we need to modify the feature space with? We will not consider feature space augmentation, but will briefly discuss it in Chapter 5.

3.3 Geometric Integration

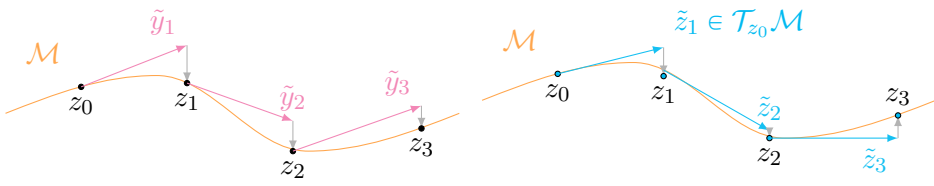
Given data $(x_i, c_i)_{i=1}^N$, where $x_i \in \mathbb{R}^{m\times n}$ or $x_i \in \mathbb{R}^{s\times m\times n}$, is either a black and white image or a colour image with s colour channels. The corresponding data point has the label $c_i \in \mathbb{R}^K$. Suppose the classification problem can be formulated as the continuous optimal control problem (3.11) subject to the constraints (3.12). We want to restrict the optimal control problem to a lower-dimensional manifold. This is the low-rank k manifold $\mathcal{M}_k^{m\times n}$ where we assume $k < m, n$,

$$\min_{\dot{z}_i \in \mathcal{T}_{z_i}\mathcal{M}_k} ||\dot{z}_i - f(y_i, u)||_F, \tag{3.13}$$

$$z(t) = U(t)S(t)V(t) \quad \text{or} \quad z(t) = \boldsymbol{\mathcal{S}} \bigtimes_{n=1}^{N} U_n, \tag{3.14}$$

where we have chosen the coordinates of z to be the Singular Value Decomposition or the Tucker Decomposition (Higher Order SVD) in the case of tensors, introduced in the previous chapter. This implies that the constraint is a matrix or tensor valued differential equation on the low-rank manifold.

The minimisation problem (3.11) is solved using the ADAM method every time we receive a new suggestion from the pair (3.12),(3.13). Therefore, the numerical approach consists of solving the pair (3.12), (3.13) to obtain a solution for (3.11). Computing the solution of the ODE (3.12) restricted to



(a) Standard projection approach.

(b) Local coordinates approach.

Figure 3.5: The two numerical geometric integration approaches we will consider.

the low-rank manifold (3.13) can be done numerically by a simple integration method to approximate the solutions (3.12), (3.13) at times $t_j = jh$ for $j = 1, \ldots, L$ when $h = T/L$. Where the time-step j, corresponds to the jth layer of the network.

Recall that components of the SVD and Tucker decomposition evolve on the Stiefel manifold. As a consequence, we are interested in integration methods which conserve invariants, in particular orthogonality of $U^{[j]} = U(t_j)$ and $V^{[j]} = V(t_j)$. Note that in our definition of the constraints (3.12) and (3.13), the invariant is only weakly inferred through being a part of the low-rank manifold, and not a property of the equation itself. We will discuss two numerical integration approaches. The first is to consider a standard projection approach where we will take a step with a general purpose method before projection with Polar or SVD. The other approach is considering local coordinates, and therefore the Dynamic Low-Rank methods.

3.3.1 Projection Methods

This approach is based on the idea from the standard projection method [29]. We apply a one-step method in Euclidean space, and then make a projection onto the embedded space, see Figure 3.5(a). We will in this section not always distinguish explicitly between SVD/HOSVD. The principles remain to control the rank and the orthogonality of the orthogonal factors. The main difference between the SVD and the HOSVD is the number of orthogonal matrices, and their dimensions. The matrix $S \in \mathrm{GL}_k(\mathbb{R})$ in SVD is a diagonal matrix and the core tensor $\boldsymbol{\mathcal{S}} \in \mathrm{GL}_r(\mathbb{R})$ in the HOSVD. We will distinguish only between them when necessary.

The obvious choice of Euclidean integrator of Equation (3.13) is the ResNet evaluation step; the Euler method. We can either apply the integrator dir-

ectly to $\dot{z}$, and project onto the low-rank manifold, or we can focus on the components of z; U, S and V^T.

Low-rank using SVD Projection

We focus on the discretisation of $\dot{z}$ and therefore only satisfy (3.14) indirectly. Assume initial data on the form $X_0 = \sum_{i=1}^{k} s_i u_i v_i^T$. Now, $x_i \in \mathcal{M}_k^{m\times n}$ is low-rank and therefore $f : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$, from equation (3.12). After each step of the Euler method, we project the intermediate step $\tilde{y}^{[j+1]}$, which we cannot guarantee to be low-rank. We ensure this by decomposing the output $\tilde{z}^{[j+1]}$ as follows

$$\begin{aligned} \tilde{z}^{[j+1]} &= z^{[j]} + hf(z^{[j]}, u), \quad z(0) = x_i, \\ USV^T &= \tilde{z}^{[j+1]}, \\ z^{[j+1]} &= \sum_{i=1}^{k} s_i u_i v_i^T, \end{aligned} \tag{3.15}$$

where $\sum_{i=1}^{k} s_i u_i v_i^T$ is the truncation of USV^T. Now we are guaranteed that $z^{[j+1]} \in \mathcal{M}_k^{m\times n}$.

This is equivalent to taking SVD to ensure that we are on the low-rank manifold at every time-step. Note that we are propagating the full matrix but with reduced rank at every time-step. So in this algorithm there is no benefit from hoped possibility of dimension reduction. Instead we add the cost of computing the SVD and matrix multiplication in every layer. Lastly, note that the procedure is easily extended to tensors and the tucker(HOSVD) Decomposition. If $\boldsymbol{\mathcal{X}}_i \in \mathbb{R}^{s\times m\times n}$ the Tucker Decomposition of $\boldsymbol{\mathcal{X}}_i$ of Tucker rank $r = [s, k, k]$, yields a tensor of the desired Tucker rank r. Lastly, let $f : \mathbb{R}^{s\times k\times k}$, and the procedure follows.

We will later refer to the network implementing the procedure (3.15) as SVD ProjectionNet.

Orthogonality via Polar Projection

Assume initial data on the form $x_i = [U_k, S_k, V_k]$, where $U_k \in \mathrm{St}(m, k), V_k \in \mathrm{St}(n, k)$ and $S_k \in GL_k(\mathbb{R})$. In this method, the main focus is on the orthogonal matrices U and V, where we let the S move freely. Each factor will be considered separately. We will first discuss the matrix case before considering the tensor case.

Let $f : \mathbb{R}^{m\times k} \to \mathbb{R}^{m\times k}$ when $m \geqslant n$. We introduce a padding to the orthogonal matrix V and the matrix S to augment the size. We add $(m-n)$

rows to V and $(m-k)$ rows to S such that

$$\hat{V} = (V^T, \underbrace{\mathbf{0}, \dots, \mathbf{0}}_{(m-n)})^T \in \mathbb{R}^{m\times k}, \quad \hat{S} = (S^T, \underbrace{\mathbf{0}, \dots, \mathbf{0}}_{(m-k)})^T \in \mathbb{R}^{m\times k}.$$

Note that this padding does not break the orthogonality of V. We now have a system of differential equations for each matrix in the decomposition

$$\begin{aligned}
\tilde{U}_k^{[j+1]} &= U_k^{[j]} + hf(U_k^{[j]}, u), \quad U_k^{[0]} = U_k, \\
S_k^{[j+1]} &= S_k^{[j]} + hf(S_k^{[j]}, u), \quad S_k^{[0]} = \hat{S}_k, \\
\tilde{V}_k^{[j+1]} &= V_k^{[j]} + hf(V_k^{[j]}, u), \quad V_k^{[0]} = \hat{V}_k, \\
U_k^{[j+1]} &= \text{proj}_{St(m,k)} \tilde{U}_k, \\
V_k^{[j+1]} &= \text{proj}_{St(m,k)} \tilde{V}_k,
\end{aligned} \tag{3.16}$$

Projecting the U and V matrices by the polar decomposition is the optimal orthogonal replacement in the Frobenius norm and is a direct result from the best approximation [38, 29]. Other projection methods can be used, i.e the QR decomposition.

The tensor case is more tricky, and there are multiple ways to set up a system. We will only present one setup here, and discuss other methods in Chapter 5. The tensor $\boldsymbol{\mathcal{X}}_i \in \mathbb{R}^{s\times m\times n}$ can be decomposed using Tucker rank $r = [s, k, k]$ into a core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{s\times k\times k}$ and three orthogonal factors $U_1 \in \mathbb{R}^{s\times k}$, $U_2 \in \mathbb{R}^{m\times k}$ and $U_3 \in \mathbb{R}^{n\times k}$. Assume $m \geqslant n, s$ and let $f : \mathbb{R}^{m\times k} \to \mathbb{R}^{m\times k}$ as in the matrix case. A similar padding is applied to each orthogonal matrix, as follows

$$\hat{U}_1 = (U_1^T, \underbrace{\mathbf{0}, \dots, \mathbf{0}}_{(m-s)})^T \in \mathbb{R}^{m\times k}, \tag{3.17}$$

$$\hat{U}_3 = (V^T, \underbrace{\mathbf{0}, \dots, \mathbf{0}}_{(m-n)})^T \in \mathbb{R}^{m\times k}. \tag{3.18}$$

For the core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{s\times k\times k}$, we unfold the tensor into a matrix. In this case, where the the first mode represents the number of colour channels, we choose a 1-mode unfolding. This creates a matrix of size $(k \times sk)$. Assuming $sk \leqslant m$, we create a padding for this new matricisised tensor as well. If $sk > m$, we need to use the tensor as guide, and pad the orthogonal matrices.

$$S = (\boldsymbol{\mathcal{S}}_{(1)}, \underbrace{\mathbf{0}, \dots, \mathbf{0}}_{(m-sk)})^T \in \mathbb{R}^{m\times k}$$

This results in the system of equations

$$\begin{aligned}
S^{[j+1]} &= S^{[j]} + hf(S^{[j]}, u), \quad S^{[0]} = S, \\
\tilde{U}_1^{[j+1]} &= U_1^{[j]} + hf(U_1^{[j]}, u), \quad U_1^{[0]} = \hat{U}_1, \\
\tilde{U}_2^{[j+1]} &= U_2^{[j]} + hf(U_2^{[j]}, u), \quad U_2^{[0]} = U_2, \\
\tilde{U}_3^{[j+1]} &= U_3^{[j]} + hf(U_3^{[j]}, u), \quad U_3^{[0]} = \hat{U}_3, \\
U_1^{[j+1]} &= \text{proj}_{St(m,k)} \tilde{U}_1, \\
U_2^{[j+1]} &= \text{proj}_{St(m,k)} \tilde{U}_2, \\
U_3^{[j+1]} &= \text{proj}_{St(m,k)} \tilde{U}_3.
\end{aligned} \tag{3.19}$$

Again, the projection operator proj_{St} could be any desirable orthogonal projection. We have chosen the Polar projection. There are many ways to compute this projection, see [39].

Throughout the network we maintain the augmented size of the matrices, S, U_i, such that they can be evaluated by the network layer f. After the final layer, before applying the classifier, we truncate the matrices and multiply the propagated factors to reassemble the image. The core tensor $\boldsymbol{\mathcal{S}}$ is folded back into a tensor, the matrices are truncated and multiplied together with the core to reassemble the image. Then we apply $\eta : \mathbb{R}^{s \times m \times n} \to K$ and softmax to normalise prediction. For the matrix case, this implies we require a layer $\eta : \mathbb{R}^{m \times n} \to K$ before applying the soft max in the matrix case. The matrix S and the orthogonal matrices U and V are also truncated back to the original size before we reassemble the image.

In this case, we propagate three matrices separately. Therefore, we get the added dimension reduction bonus. Also, by ensuring that the matrices U and V are orthogonal and of dimension (m, k) we also ensure that we stay on the low-rank manifold as their product has rank k. But this time, the low-rank is only implicit and only formed after the last layer L. In this algorithm, no restrictions were put on the diagonal Σ matrix. The main reason for this was that we wanted to focus on the orthogonality constraints, but also that we wanted an algorithm which is more similar to the Dynamic Low-Rank algorithm we will see in the next section. It is possible to put restrictions on the Σ quite easily by doing the factorisation $S = \tilde{S}\tilde{S}^T/2$, but this has not been explored.

We will later refer to the network implementing the procedure (3.16) or (3.19) as ProjectionNet.

Advantages and Disadvantages

The cost of computing the SVD and the polar decomposition on high-dimensional matrices is expensive, even on GPUs. This is a major drawback for these "hard constraint" projection method. However, this is by far the easiest to implement. These methods require fewer matrix multiplications than the Dynamic Low-Rank methods in the next section.

The SVD ProjectionNet, equation (3.15), will only be a theoretical experiment as it does not lead to a reduced model. It is mainly interesting to investigate how the performance is affected by truncating the output to low-rank. We will also test this methods robustness to adversarial attacks, to see if there are any benefits from preserving low-rank.

For the ProjectionNet, the padding of the tensors and matrices is done only once, when setting up the data set. Lastly, the factors of the decomposition are not dependent on each other. The dependent approach will be considered in the next section.

3.3.2 Lie Group Integrators

We now return to the optimal control problem (3.11) subject to the ODE constraint (3.12) restricted to the low-rank manifold (3.13). We now want to use local coordinates to solve the ODEs.

One way to choose local coordinates, which does not require a specific structure of the manifold, is to use tangent space parametrisation. We can think of this as propagating in the direction of the tangent space, and then projecting down onto the manifold, as seen in Figure 3.5(b). Recall from the section on dynamic low-rank approximation in the previous chapter that this is the approach taken to derive the system of equations for the evolution of the low-rank approximation. We will now revisit the last two methods discussed previously; the dynamic low-rank approximation and the dynamic tensor approximation. The systems will be solved and discretised using a simple first-order scheme.

Dynamic Low-Rank Approximation

We are interested in the low-rank approximation to the matrix valued differential equation (3.13), which we approximate by the neural network $F(Y_i, u_i) = \dot{Y}$. This yields the minimum-defect low-rank approximation (3.13) with coordinates (3.14). Recall the system om non-linear ODEs from Theorem 2.1.

We restate the equations for the components here

$$\begin{aligned}\dot{S} &= U^T\dot{Y}V \\ \dot{U} &= (I - UU^T)\dot{Y}VS^{-1} \\ \dot{V} &= (I - VV^T)\dot{Y}^TUS^{-1}\end{aligned} \tag{3.20}$$

We have chosen the ease the notation by writing $\dot{Y}$ instead of $F(Y, u)$, but in the implementation, we evaluate $F(Y, u)$.

The system (3.20) can be rewritten on the form

$$\begin{aligned}\dot{S} &= U^T\dot{Y}V, \\ \dot{U} &= (F_UU^T - UF_U^T)U, \quad F_U := (I - UU^T)\dot{Y}VS^{-1}, \\ \dot{V} &= (F_VV^T - VF_V^T)V, \quad F_V := (I - VV^T)\dot{Y}^TUS^{-T},\end{aligned}$$

such that we can apply the Cayley transform instead of the exponential map for efficiency. The Cayley transformation can be defined as follows

$$\text{cay} : \mathbb{R}^{m\times m} \to \mathbb{R}^{m\times m}, \quad \text{cay}(B) = \left(I - \frac{1}{2}B\right)^{-1}\left(I + \frac{1}{2}B\right).$$

It is well known that if B is a skew-symmetric $m \times m$ matrix then $\text{cay}(B)$ is an orthogonal matrix, see Appendix A for more details.

A simple first order integration method for the system of ODEs can take the form of a forward (Lie-)Euler method for the system. This method is given by $y^{n+1} = \exp(hF_{y_n})y_n$, where F_{y_n} is the tangent vector field at the point y_n. We use the Lie-Euler methods for the equations evolving on the Stiefel manifold. The diagonal matrix $S \in \text{GL}_k$ is integrated using classic Euler. This yields the discrete system

$$\begin{aligned}S^{[j+1]} &= S^{[j]} + h\,U^{[j]T}\dot{Y}^{[j]}V^{[j]}, \\ U^{[j+1]} &= \text{cay}\left(h(F_{U^{[j]}}U^{[j]T} - U^{[j]}F_{U^{[j]}}^T)\right)U^{[j]}, \\ F_{U^{[j+1]}} &:= (I - U^{[j]}U^{[j]T})\dot{Y}^{[j]}V^{[j]}S^{[j](-1)}, \\ V^{[j+1]} &= \text{cay}\left(h(F_{V^{[j]}}V^{[j]T} - V^{[j]}F_{V^{[j]}}^T)\right)V^{[j]}, \\ F_{V^{[j+1]}} &:= (I - V^{[j]}V^{[j]T})\dot{Y}^{[j]T}U^{[j]}S^{[j](-T)}.\end{aligned} \tag{3.21}$$

We have chosen this somewhat cumbersome notation to ease the extension to tensors in the next subsection, and also keep consistency.

Dynamic Tensor Approximation

Lastly, we are interested in the low-rank tensor approximation to the tensor valued differential equation (3.13) with the coordinates (3.14). In the case of tensor valued differential equations on the low-rank- and Stiefel manifolds, we apply the same first order time integration method as in the matrix case. Recall the evolution of the Tucker decomposition from the section on dynamic tensor approximation in the previous chapter

$$\begin{aligned} \dot{\boldsymbol{\mathcal{S}}} &= \dot{\boldsymbol{\mathcal{Y}}} \bigtimes_{\boldsymbol{n=1}}^{\boldsymbol{N}} U_n^T \\ \dot{U}_n &= (I - U_n U_n^T)[\dot{\boldsymbol{\mathcal{Y}}} \bigtimes_{\boldsymbol{n\neq k}} U_k^T]_{(n)} S_{(n)}^{\dagger}, \end{aligned} \tag{3.22}$$

with the pseudo-inverse $S_{(n)}^{\dagger} = S_{(n)}^T (S_{(n)} S_{(n)}^T)^{-1}$ of the n-mode unfolding $S_{(n)}$ of $\boldsymbol{\mathcal{S}}$.

We present the solutions for mode N tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, for simplicity. The system (3.22) can be rewritten on the form

$$\begin{aligned} \dot{S} &= \dot{\boldsymbol{\mathcal{Y}}} \bigtimes_{n=1}^{N} U_n^T, \\ \dot{U}_n &= (F_{U_n} {U_n}^T - U_n F_{U_n}^T) U, \\ F_{U_n} &:= (I - U_n U_n^T)[\dot{\boldsymbol{\mathcal{Y}}} \bigtimes_{k\neq n} U_k^T]_{(n)} S_{(n)}^{\dagger}, \end{aligned} \tag{3.23}$$

where we again write $\dot{\boldsymbol{\mathcal{Y}}} = F(\boldsymbol{\mathcal{Y}}, u)$, $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{S}} \times_{n=1}^{N} U_n$.

We discretise the equations (3.23) and arrive at the system

$$\begin{aligned} S^{[j+1]} &= S^{[j]} + h \left(\dot{\boldsymbol{\mathcal{Y}}}^{[j]} \bigtimes_{n=1}^{N} U^{[j]T} \right), \\ U_n^{[j+1]} &= \operatorname{cay}\left(h(F_{U_n^{[j]}} U_n^{[j]T} - U_n^{[j]} F_{U_n^{[j]}}^T) \right) U_n^{[j]}, \\ F_{U_j} &:= (I - U_n^{[j]} U_n^{[j]T})[\dot{\boldsymbol{\mathcal{Y}}}^{[j]} \bigtimes_{k\neq n} U_k^{[j]T}]_{(n)} S_{(n)}^{[j]\dagger}, \end{aligned} \tag{3.24}$$

where we have N orthogonal matrices, one for each mode and one tensor equation. Note that in this case the N is the total number of n-mode factors in the tucker decomposition, and not the size of the data set in the neural network.

We will later refer to the network implementing the procedure (3.21) or (3.24) as DynamicNet.

3.3.3 Advantages and Disadvantages

Compared to the SVD projection method, solving these differential equations are cheaper than computing the SVD at each time step. However, the many matrix multiplications are also very expensive. Both the dynamic low rank and dynamic tensor approximations do not augment the size tensors or the matrices. However, in order to evaluate the vector field F we need to reconstruct the images st every time step. This is very time-consuming. Furthermore, the differential equations give smooth solutions evolving on the manifold, however, it is unclear which benefits this might have. One could suspect that manifold traversal makes more sense on smooth manifolds. Lastly, these methods are not yet order-reducing, but it has potential. And especially for tensors and the Tucker decomposition one can potentially reduce the order by a large factor if the image tensor is large.

3.4 Adversarial Attacks

Machine learning models have proven to be extremely effective at certain tasks. As these models enter the real world, one aspect that is often overlooked is the security and robustness of the models, especially in the face of an adversary who wishes to fool the model. Awareness of the security vulnerabilities of these machine learning models, will in the future, only become more and more important. In particular, neural networks are vulnerable to *adversarial attacks*. Two types of adversarial attacks are adversarial noise and adversarial rotation. We can carefully create noise such that when added to the image, the network will misclassify, while the naked human eye cannot see any difference. Furthermore, given a specific rotation to an image reproduce the same misclassification of the algorithm, while a human understands that the content of the image is only rotated. Adding such imperceptible perturbations to an image can cause drastically different model performance [45].

There exist many methods that seek to fool our network, each with a different goal and assumption of the attackers' knowledge. *Black box models* assume the attacker only has access to inputs and outputs of the model, and in *white-box models*, where the attacker has access to everything; the data and network structure and trained parameters. There are different goals for the attack; *misclassification* and *source/target misclassification*. If the adversary only cares for the prediction to be wrong, the goal is simply

misclassification. On the other hand, if the adversary wants the perturbed image to be classified within a specific class, then it specifies as source/target misclassification [45, 80].

There exist a plethora of algorithms with different assumptions and accesses. We will discuss a very popular white-box model with the goal of misclassification called the Fast Gradient Sign Method (FGSM). See [80] for an extensive list of methods and properties and also suggestions for defence strategies. In this paper, the authors also shed light on the need to establish methodologies for robustness evaluations and benchmark platforms for comparison and reproducibility. As deep learning models are entering real-world applications, this research is becoming more and more important to understand and prevent attackers from hacking our models [58]. One can only imagine the damage that can be done to self-driving cars and the traffic system.

3.4.1 Fast Gradient Sign Method (FGSM)

Let x be the input data, and y be the output. By linearising the cost function $J(\theta, x, y)$ around the current parameters θ, we obtain the max-norm optimal constrained perturbation of

$$\xi = \epsilon \text{sign}(\nabla_x J(\theta, x, y)). \tag{3.25}$$

See [25] for details. We can add this perturbation to the original image to generate an adversary input

$$\tilde{x} = x + \xi. \tag{3.26}$$

The algorithm starts with an image the network correctly classified. Then it adjusts the input to maximise the loss based on the computed gradient. This adjusted input is added on top of the original image, creating a new perturbed image. For larger and larger ϵ, this adjusted input becomes more and more visible in the perturbed images. The adversarial attack will be easier to detect for a human, but more effective on the network. This method will be used on fully trained models to see how *robust* they are to subtle changes in input [25].

3.5 Closing Remarks

There are other ways to control the parameters of the network. Many papers have shown great results by controlling the weights of the network [61, 31, 42, 1, 3]. This might be two sides of the same coin, and therefore, the

connection might be worth exploring. Also, in the analysis of convergence of the ResNet proposed in [75], they used regularisation of the weights as a necessary condition for convergence. Finding similarities between restricting the evolution of the network and the evolution of the internal weight could yield another approach to the convergence of ResNets.

We can in many ways relate Adversarial Attacks (3.26), (3.25) to stability of ODEs [72]. In the recent paper by Kang et.al [50], the authors show great improvement in adversarial robustness using Lyapunov stability in Neural ODEs, tested with the FGSM. Furthermore, one can also discuss the well-posedness of the problem, see [27].

By imposing weight regularity, Lipshitz regularity, orthogonality one might preserve mathematical properties such as convergence towards the analytical solution, generalisability, stability and well-posedness. We can design networks and methods which preserves a problem specific structure, see [10].

Let us turn back to the example with handwritten digits. After training the network, we can give the network a random picture of white noise. One would hope that the network would gove an uncertain prediction, but often this is not the case. One way to explain this behaviour is that the network is not expecting this type of input. This is essentially the wrong classifier for the problem at hand. The example illustrates how inflexible neural networks can be and that they are good at particular tasks. They do not know what the digit "actually" looks like but have just learned ways to reproduce the correct answer. We call this phenomenon memorisation [44].

Over-fitting is another such problem. Here, the network performs well on the training data set but does not perform as well on validation data. The model does not generalise to new data and we can suspect the network has focused too heavily on noise and unimportant patterns in the training data. If we cannot see an improvement in the accuracy on the validation data during the training process, we can terminate the training by early stopping. A related technique to regularisation is *dropout*. Here, a random neuron is removed with probability p during optimisation.

Another obstacle is the notorious *vanishing-* and *exploding gradient* problem. In the papers from 1994 [5] and 2010 [22], the authors detail how and why the gradient descent method in very deep networks will impede convergence. Here, the networks' weights are updated with an error proportional to the current weights in each iteration of training. Thus, in some cases, the gradient will be vanishingly small, such that the weight will not change. In the

case of exploding gradients, the error grows, leading the gradient to increase exponentially. In both, the model is incapable of learning efficiently, and in the latter, the model is unstable. Solutions have been proposed and listed in [35]. One such method is gradient clipping, where we ensure that the norm of the gradient does not exceed a given threshold. It has been shown that ResNets do not have the problem of vanishing- and exploding gradients due to the residual connection.

Chapter 4

Numerical Experiments and Results

It's all to do with the training:
you can do a lot if you're properly trained.
– Queen Elizabeth II

In this chapter we present numerical experiments. In the first section we will present the setup. In particular details on implementation and parameters. In the second section we will investigate the data sets to see whether we can find indications of a lower-dimensional manifold and choose a truncation for the SVD and Tucker Decompositions. In the third section we will demonstrate the need for specific methods to restrain the rank though the network. In the forth section we will test our developed methods on a few chosen data sets. Lastly, in section five we will investigate the robustness of the models. In particular, the main result of this thesis which is that our methods are more robust to adversarial FGS attacks, especially for deep networks.

4.1 Setup

The code[1] used for the experiments has been implemented in Python, where the machine learning architecture and networks have been built using PyTorch [67] and NumPy [32]. The code setup was inspired by the code from [21]. When extending the code to tensors, we have relied heavily upon the framework provided by TensorLy [55], which also contains an interesting

[1] https://github.com/Camilbk/Dynamic-Low-Rank-Network

framework for training weight tensors built upon PyTorch. All plots have been produced using Matplotlib [43].

We construct a neural network as follows; we choose a fully connected structure of L linear layers on the form $f(y^{[l]}) = \sigma(W^{[l]}y^{[l]} + b^{[l]})$. We have chosen $\sigma := \text{ReLu}$ for all the experiments and networks. The width of each layer is constant and defined by the dimension of the input layer $\mathcal{X}_0$. To reduce the dimension of the final layer to the dimension of output layer, we have chosen a linear classifier, which is complemented by softmax. The loss criterion is categorical cross entropy and the optimisation method is set to ADAM. We have no weight decay.

In our implementation of the neural network, a few decisions have been made. First, we have decided to keep the width constant. The narrowing and expansion of the width of the network might enhance the performance. However, benchmark performance is not the goal of this thesis. When choosing layers of constant width, this is to ensure that we keep most variables constant. In that way we can be more certain about which behaviours stem from what. The same argument can be used for convolutions. We know that convolutional layers enhance the performance of the classification. We choose a simple and more explainable model to make sure to remain in control of most of the behaviour. Also, we have not augmented the feature space, even though we know this increases the flexibility and degrees of freedom of the network.

We have chosen four data sets to test our networks on. Two of them are black and white images, and therefore chosen for testing the algorithms on matrices. These are the MNIST [13] and FashionMNIST [79] data sets. Both of these contain (28×28)-size images. MNIST is known to be easy to learn for networks. To test the networks on tensors, we have chosen two coloured data sets the: CIFAR10 [56] and Street View House Numbers (SVHN) [65]. Both CIFAR10 and SVHN are $(3 \times 32 \times 32)$. SVHN is similar to MNIST in many ways as it contains images of digits. But it is a coloured data set, which adds another dimension of difficulty. It is also more noisy compared to MNIST, and contains pieces of other numbers as well. Three samples from each data set and how they respond to SVD and Tucker Decomposition is shown in Appendices A.2, A.3, A.4 and A.5. For shuffling and dividing up the training and validation data sets we have used the framework provided by PyTorch.

The only parameters we can and will control during training are the size of the training data set N, the size of the validation data set V, the batch

size, number of epochs and depth L. In most cases we have chosen relatively small training and validation data sets, $N = 1500, V = 1500$, to ease the computation time. For all data sets we have chosen a batch size between 5 and 30. The number of epochs have been chosen in the specific case to ensure that the methods have converged. Here we will test our methods on $L = 10$ and $L = 100$, to see how the networks respond to more trainable depth parameters. We will often denote the depth of the networks by an acronym, i.e. ResNet-10. When plotting accuracy, the solid lines represent the training accuracy, and the dotted lines represent the validation. If we run with other parameters than stated in this section, we will explicitly state the parameters and their value.

We have in total four networks, presented in the previous chapters.

1. The standard ResNet, which is defined by the propagation between the layers as seen in equation (3.8). This network does not have any additional properties, and is implemented for tensors and matrices.

Then we have discussed the Projection Networks. These networks are a natural extension of the ResNet formulation.

2. The SVD Projection is defined by the propagation as seen in equation (3.15) and where, after a propagation step, we take the SVD/HOSVD of the output at every layer.

3. ProjectionNet is defined by the propagation as seen in equation (3.16) for matrices, and also (3.19) for tensors. In this case we propagate the factors of the decomposition instead of the full image. After a ResNet step, we ensure that the matrices are orthogonal via a polar projection.

Lastly, we have DynamicNet which based on a continous formulation of the SVD.

4. DynamicNet is based on the dynamic low-rank approximation formulation for matrices, and the dynamic tensor approximation for tensors, see equations (3.21) and (3.24) respectively. DynamicNet describes the evolution of the factors of the decomposition on the manifold in a continuous setting.

In DynamicNet, the factors are dependent on each other. This is not the case in ProjectionNet. Another difference between these two methods is therefore

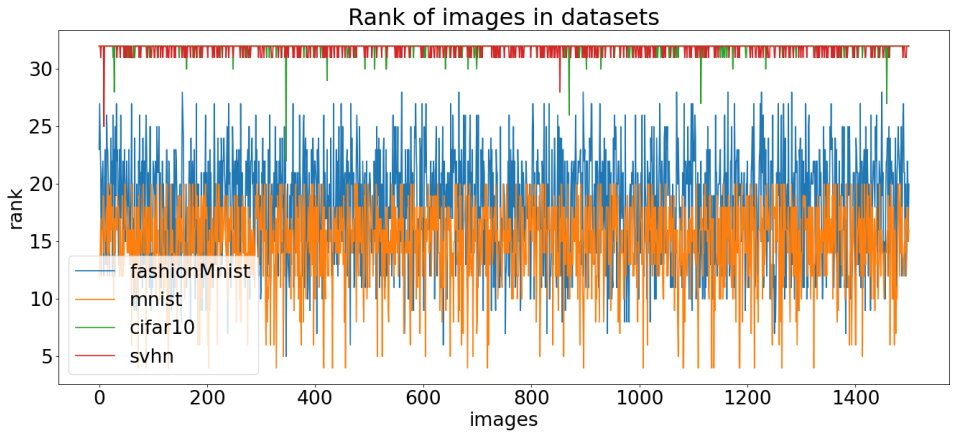


Figure 4.1: Rank of the matrix plotted for each image in data set. Note that CIFAR10 and SVHN are grayscale.

evaluation of the vector field, the linear layer $f(y^{[l]})$. The ProjectionNet evaluates the components of the decomposition, while the DynamicNet evaluates the restored image. Therefore, a ProjectionNet has the major advantage of model order reduction. DynamicNet not only has to perform many matrix and tensor operations, but also has to restore the image so that it can be evaluated in the layer. For the ProjectionNet, the new input size is only $\approx 11\%$ of the original matrix size, we have achieved a similar compression with $\approx 1\%$ of the original tensor size.

4.2 Investigating the data sets

At the start of Chapter 1 we claimed that one way to investigate whether a data set naturally belongs to a lower dimensional manifold is to investigate the rank of the images. Therefore, we will investigate the rank of the images in the various data sets.

As we do not have an efficient algorithm for calculating the rank of a tensor, which is comparable to the rank of a matrix, we will first convert the coloured images into grayscale. In this way, the three colour channels are projected down to one dimension, such that the tensors are now matrices. The variation of ranks though the data sets can be seen in Figure 4.1.

Firstly; notice that FashionMNIST has a higher average rank than MNIST. Note that some images have very low rank, especially in MNIST, and none of the images are full rank. It is clear from Figure 4.1 that the rank of grayscale CIFAR10 and SVHN have much less variation of ranks across the

data set. The rank of the images are full or almost full in all cases. We can suspect that the projection of the colour channels has increased the rank of the matrices such that they are artificially high. Especially when looking at samples from SVHN, Appendix A.5, the images have more in common with MNIST, Appendix A.2, such as sparsity. Furthermore, by looking at how much the ranks of images vary in the other two matrix data sets, we can suspect that the rank of the grayscale image does not show the complete picture. Therefore, it is a better measure to investigate the error in the Tucker Decomposition, to get an idea of what the rank truly is. But first we investigate the distribution of the singular values. With these experiments we have build a certain confidence that our data sets live in a low-dimensional manifold.

We shift our focus towards investigating the singular values of the data sets, see Figure 4.2. When looking at MNIST and FashionMNISTs singular values on the top row. The leading singular values have similar spreads. If we choose the $k = 3$ leading singular values in MNIST and FashionMNIST we know from Eckart-Young-Mirskys theorem that the larges singular value omitted gives the error in the low rank approximation (2.3.1). Therefore, we end up with an error $||A - A_k||_2 = \sigma_{k+1} \in [0, 4]$ for both data sets. It seems also that choosing only two terms in the truncated singular value decomposition gives a reasonably good approximation, but we will use 3 terms for good measure, see Appendix A.2, A.3. Maybe more surprisingly, the grayscale CIFAR10 and SVHN have a large leading singular value. It seems from the bottom row of Figure 4.2 that there are not many significant values needed to get a good approximation of the grayscale images. This is slighty surprising in light of to the results in Figure 4.1. In grayscale, we can easily pick two or three singular values and get a lower error in the low-rank approximation of the grayscale than in the approximation for MNIST and FashionMNIST.

If we turn our attention to the Tucker Decomposition of the tensors, see Figure 4.3(a) and Figure 4.3(b), we notice similar results here. In these Figures, we have plotted the error, $||Y - Y_k||_2$, between the original tensor and the Tucker Decomposition of Tucker rank $r = [3, k, k]$ for different ks. Not surprising, the Tucker Decomposition using $k = 3$ gives the highest error in both data sets. Choosing $k = 9$ in the case of CIFAR10 gives us a maximum error of around 4. In the case of SVHN we can choose $k = 5$ and get a similar error, but we choose $k = 9$ as well for simplicity.

We have now chosen a truncation for each data set, based on the error in the singular values and error in the decompositions. To summarise, we have

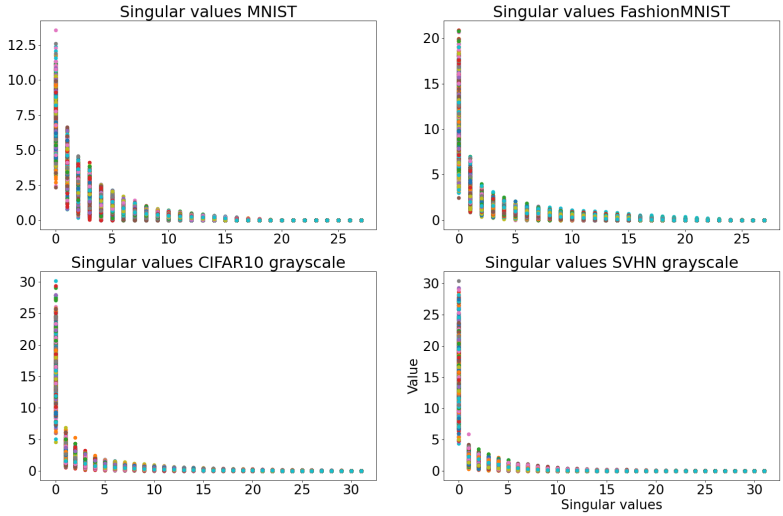


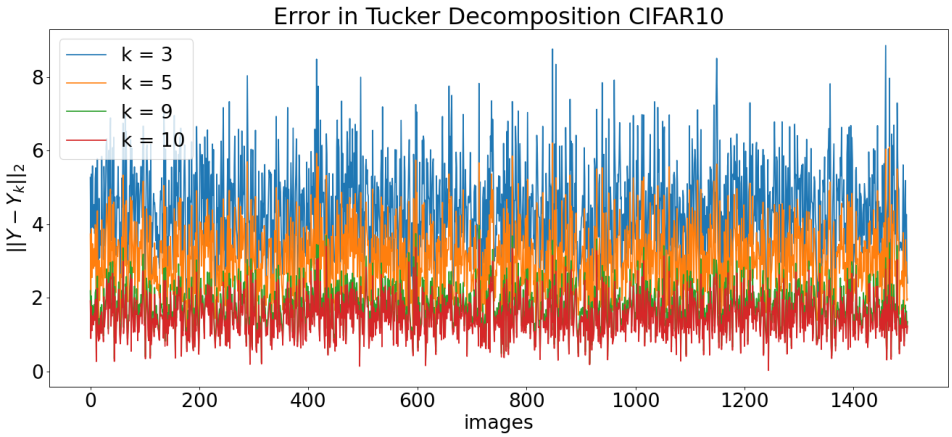
Figure 4.2: The singular values of each matrix in data sets plotted as points.

achieved a maximum error of around four in all cases. With the truncated SVD for MNIST and Fashion MNIST with $k = 3$. The Tucker Decomposition of CIFAR10 and SVHN with Tucker rank $r = [3, k, k]$ where $k = 9$. We could be more rigorous when it comes to estimate a suitable k. We choose the rank based of visual impressions in the data sets, as seen in the Appendix, and also on error. We will continue the discussion in the next chapter.

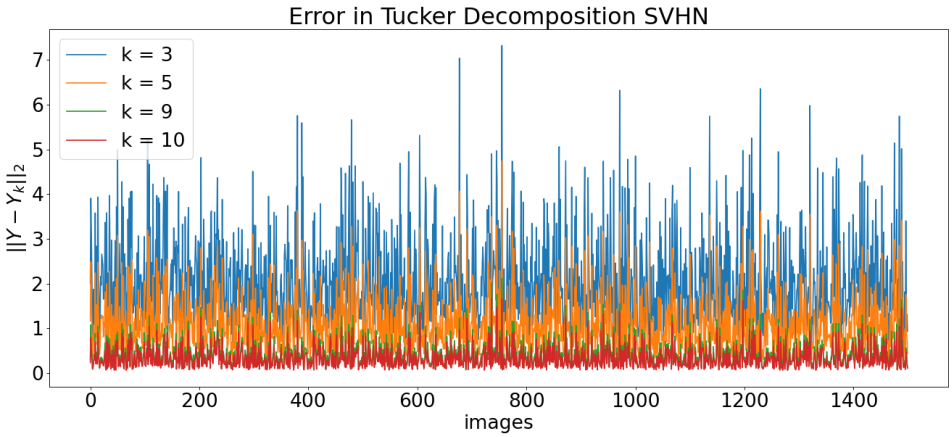
4.3 Rank Evolution

In order to study the images on the low rank manifold we need to constrain the rank using the methods proposed in the previous chapters. To illustrate why we need these specific methods we will perform an experiment to see how the ResNet is altering the rank of input.

We choose two standard ResNets. For one network we train and test on original and unperturbed data, $X_0 \in \mathbb{R}^{m \times n}$. For the second ResNet, we input the truncated SVD, $X_0 = \sum_{i=0}^{k=3} \sigma_i u_i v_i^T = A_k \in \mathcal{M}_k^{m \times n}$. So as input, one network starts with a full-rank input, and the other starts with a low-rank. As previously mentioned, we do not have an efficient algorithm to compute a comparable rank of tensors to the matrix rank. We will therefore only do this experiment on MNIST and FashionMNIST. The accuracy of



(a) Error in CIFAR10 images using various Tucker Decompositions of Tucker rank $r = [3, k, k]$.



(b) Error in SVHN images using various Tucker Decompositions of Tucker rank $r = [3, k, k]$.

the trained networks and the rank through the layers can be seen in Figure 4.4.

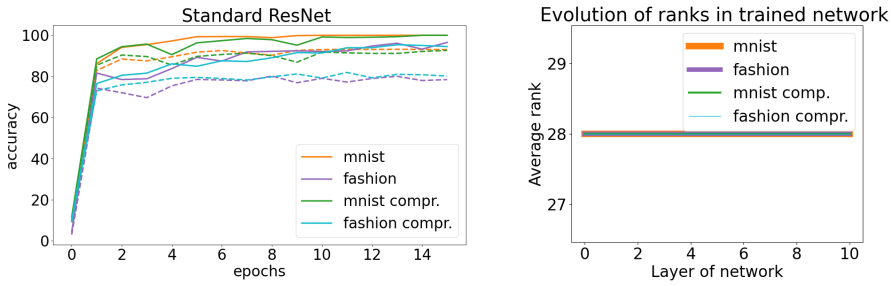


Figure 4.4: *Left:* Convergence of ResNets-10 with unperturbed input compared to ResNets-10 compressed with truncated SVD as input on MNIST and FashionMNIST. *Right:* The evolution of the ranks of the output through the layers of the trained networks.

	Original		Compressed	
	MNIST	FashionMNIST	MNIST	FashionMNIST
Training accuracy	100.0	96.53	100.0	95.40
Validation Accuracy	93.2	80.20	92.66	81.93

Table 4.1: Table of performance results for Standard ResNet with different input.

We ran 15 epochs, as this is sufficient to get an idea of how the rank evolves though-out the networks. This is also sufficient to get an idea whether or not there is a difference in performance when training on compressed vs. original data. As we can see from Figure 4.4, the methods converge nicely. What is interesting from both Figure 4.4, and the performance results shown in Table 4.1, is that the performance does not seem to deteriorate using truncated input on these experiments. Furthermore, notice from Figure 4.4 how the ranks of the data remain constant through the network. Notice, in particular, that the rank of the matrix in the input layer is also 28. The reason for this is that we apply weights to the input matrix. Due to PyTorchs' linear layer structure, we are essentially flattening the images and premultiplying them with a matrix of larger size. It seems this increases the rank of the truncated input image to full rank. This is true for all layers in the network, and for this reason we need to restrict either the output after we modify with the ResNet step or we need to constrain the weights and bias. We have chosen to go for the first option and will now implement and perform experiments to keep the rank low throughout the network.

4.4 Networks on Low-Rank and Stiefel manifolds

Now, we will continue with experiments restricting the rank and orthogonality of the matrices and tensors throughout the training of the network. First we will investigate only restricting the networks to the low-rank manifold using SVD/HOSVD, and then we move to investigate the ProjectionNet and DynamicNet, which also preserve orthogonality. We will split each section into two, the first one focusing on the matrix case, and the second focusing on the tensor case.

4.4.1 Low-Rank

In this section we perform experiments that restricts the output of each layer to the low-rank manifold. We continue with the two methods from last section; ResNet and Compressed ResNet, where compressed ResNet signifies that the initial input images have been compressed via a truncated SVD. We also include the SVD Projection methods, (3.15). This ensures that the output of each layer is on the low-rank manifold. We will call this method ResNet restricted, meaning that its evolution is restricted to the low-rank manifold.

MNIST vs FashionMNIST We compare the performance of the three methods on MNIST and FashionMNIST. The convergence plots for these methods can be seen in Figure 4.5. From these convergence plots, we see

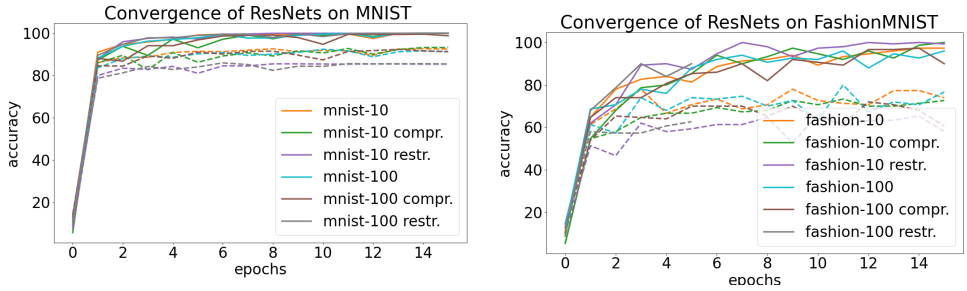


Figure 4.5: Convergence of ResNets on MNIST and FashionMNIST for networks of depths $L = 10$ and $L = 100$.

that the methods restricted to the low-rank manifold are performing worse that the other two networks. Note that in the plot to the right of Figure 4.5, "fashion-100 compressed" does not perform more than 5 epochs. This is due to the fact that performing SVD in every layer will in some cases be nu-

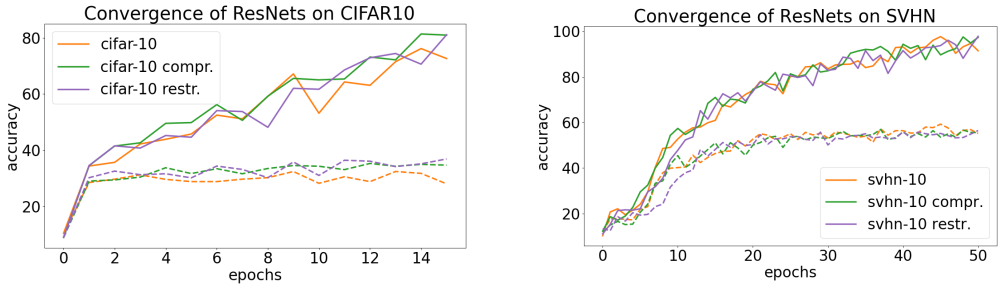
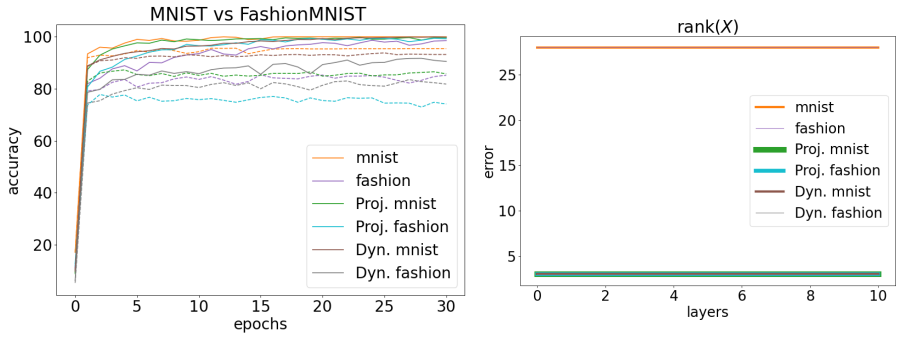


Figure 4.6: Convergence of ResNets on CIFAR10 and SVHN for networks of depths $L = 10$.

merically unstable[2]. Even though all matrices admit an SVD, the numerical procedure to compute the SVD fails if there are too many repeated singular values. Exactly why there is a difference between MNIST and FashionMNIST is not known, but MNIST has a reputation for being a forgiving data set. We also had similar experience with fashion-10 compressed, but in this particular run it did not fail to converge. It is clear that this method is not a good choice as we cannot rely on a method that in many cases fail to converge. But apart from the method being numerically unstable, it seems that the accuracy deteriorates only slightly when we force SVD at every layer. Lastly, the experiment was performed with $N = 1500$ and $V = 1500$. For this reason, we could expect that the results on Fashion could be improved slightly by training and testing on larger data sizes. However, the instability of the restricted methods are more visible with larger N, V so for this reason, this was not done.

CIFAR vs SVHN We will compare the performance of the three methods on CIFAR and SVHN. The convergence plots for these methods can be seen in Figure 4.6. In Figure 4.6, it is clear that none of the methods have sufficient convergence on the data set. It is clear that the standard ResNet network is not able to the learn patterns for sufficient classification on these data sets, and the networks are over-fitting. However, on a positive note, both the networks with compressed initial images, and compressed initial images restricted to the low-rank manifold perform as well as the standard ResNet. For SVHN the results are better, and the validation accuracy is following the training accuracy up to around 60%. We have chosen not to run deep networks on CIFAR10 as the computation is very time consuming.

[2]https://pytorch.org/docs/stable/generated/torch.linalg.svd.html



(a) (b)

Figure 4.7: *Left:* Convergence of the three methods; ResNet, ProjectionNet (Proj.) and DynamicNet (Dyn.) on both data sets. *Right:* Evolution of the average rank of the output for each layer in the trained networks.

4.4.2 Low-Rank and Stiefel

We will now investigate the methods which also evolve on the Stiefel manifold. In both sections we will train the data sets on the standard ResNet, the ProjectionNet, and lastly the DynamicNet.

MNIST and Fashion MNIST For the standard ResNet we use original unperturbed data $X_0 \in \mathbb{R}^{m \times n}$. For the ProjectionNet we apply an SVD to every image matrix and sort them in a vector, $X_0 = [U, \Sigma, V^T]$. We have chosen to go for a large training data set $N = 5000$ and validation data set $V = 1500$ in this experiment.

The input to the ResNet is, as previously mentioned, the full image. This implies weight matrices of size $W_i^{[l]} \in \mathbb{R}^{m \times n}$. The ProjectionNet has only weight and bias matrices of size $W_i^{[l]} \in \mathbb{R}^{m \times k}$, which results in fewer trainable parameters. In DynamicNet we, unfortunately, need to evaluate the vector field which we are approximating, which leads us to also having weight matrices in $W_i^{[l]} \in \mathbb{R}^{m \times n}$. Then we are ready to train and compare the networks, and the convergence can be seen in Figure 4.7(a). It is clear from Figure 4.7(a) that all the methods converge quite well. The DynamicNet has slower convergence, but the training accuracy is approaching 100%. It seems that the ResNets and the ProjectionNets have similar training convergence. However, from the plot, its seems like the ProjectionNets' validation accuracy have stabilised at quite a low accuracy. Significantly lower than DynamicNet.

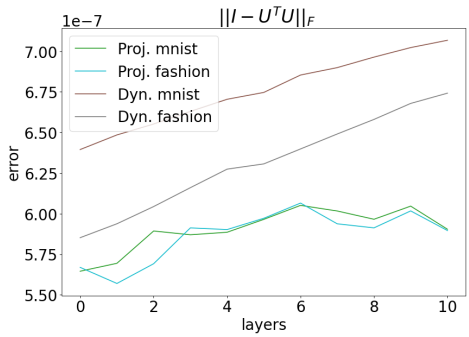
	ResNet		ProjectionNet		DynamicNet	
	MNIST	Fashion	MNIST	Fashion	MNIST	Fashion
Training Accuracy	100.0	98.66	99.80	99.44	99.96	91.72
Validation Accuracy	95.59	85.40	87.26	77.80	93.80	83.53
Time	1063s	1109s	974s	963s	1900s	1990s
Step size h	0.0784	0.0565	0.1400	0.1373	0.001	0.001

Table 4.2: Table of maximum training and validation accuracy for the different networks on MNIST and FashionMNIST. $N = 5000, V = 1500$, and the time taken on 30 epochs and the step size h for each method.

In Table 4.2, we see the maximum training and validation accuracy for the networks. The performance of DynamicNet is almost as good as the performance of ResNet. The same cannot be said for ProjectionNet, where we have almost 10% difference in the validation accuracy. Lastly, note the time taken for the networks to perform 30 epochs of these algorithms. The ProjectionNet beats the ResNet significantly, so there is definitely something to gain computationally by using ProjectionNet. It is worth noting that these time experiments have not been rigorously done, and the time taken for the networks should be taken with a grain of salt. However, they give an indication of the computational costs of the algorithms. Lastly, the algorithms have not been implemented in the most efficient way, as the focus was readability and correctness rather than speed.

We also show numerical evidence of the orthogonality of U and V in both ProjectionNet and DynamicNet, see Figure 4.8(a) and 4.8(b), and the rank of the output at each layer, see Figure 4.7. From the orthogonality plots, the matrices U and V remain orthogonal throughout the network. Even though the orthogonality produced by the projection network does not vary by a lot, it seems significant compared to the smooth variation produced by the DynamicNet. The algorithms produce outputs on the low-rank manifold, as is clear from Figure 4.7. The rank of the output from ProjectionNet is computed by multiplying U, S and V of the trained network. Only the standard ResNet has full rank output after each layer.

From table 4.2 we can see the trained step-sizes h from ResNet and ProjectionNet and the constant step size for DynamicNet. The step size is the same for all layers. We have chosen not to train the step size of DynamicNet, as it seemed to produce instabilities during the training process in most cases. Therefore, we reduced it to $0.1/L$. The reasons behind the instabilities have not been investigated. In the case of ResNet and ProjectionNet we chose to initialise the step size as $1/L$. The final trained step size is very different



(a) Average orthogonality error in U

(b) Average orthogonality error in V

Figure 4.8: Orthogonality error in U and V for ProjectionNet and DynamicNet on MNIST and FashionMNIST for each layer of the trained network.

	ResNet100		ProjectionNet100		DynamicNet100	
	MNIST	Fashion	MNIST	Fashion	MNIST	Fashion
Training Accuracy	99.98	97.70	99.70	99.62	99.98	94.73
Validation Accuracy	95.47	84.27	86.70	77.13	93.73	82.87
$\approx$ Time	2800s	2900s	2500s	2500s	7400s	7900s
step size h	0.0173	0.0190	0.0411	0.0551	0.001	0.001

Table 4.3: Table of best training and validation accuracy for the different deep networks, $L = 100$, on MNIST and FashionMNIST, and the time taken on 30 epochs. The size of the training data set was increased to 5000.

for the two networks, but similar for the different data sets. The different step sizes illustrate again how different the networks behave. We do these measurements on the trained networks. Therefore it is not clear if there is a difference between the errors in orthogonality during training and after.

Lastly, we provide a summary of results for trained networks on $L = 100$, see Table 4.3. There is a slight improvement when increasing the depth of the network, as expected. Plots of the orthogonality errors are very similar to the shallow networks, and these plots along with the accuracy and rank evolution can be found in Appendix A.6.

CIFAR10 and SVHN We start by preparing the data sets. For the standard ResNet we use original unperturbed data $X_0 \in \mathbb{R}^{3\times m\times n}$. For the ProjectionNet and DynamicNet we input a vector of the Tucker Decomposition $X_0 = [\boldsymbol{\mathcal{S}}, U_1, U_2, U_3]$ of Tucker rank $r = [3, k, k]$ where we previously chose $k = 9$. This vector contains a tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{3\times k\times k}$, and three matrices

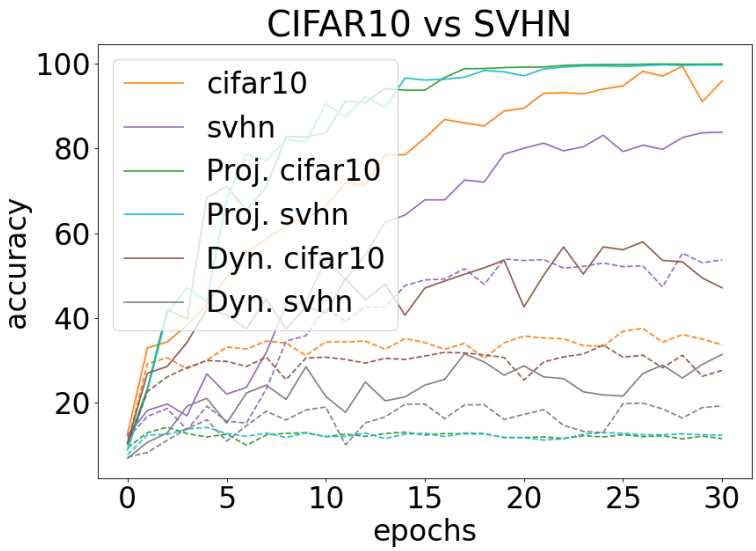


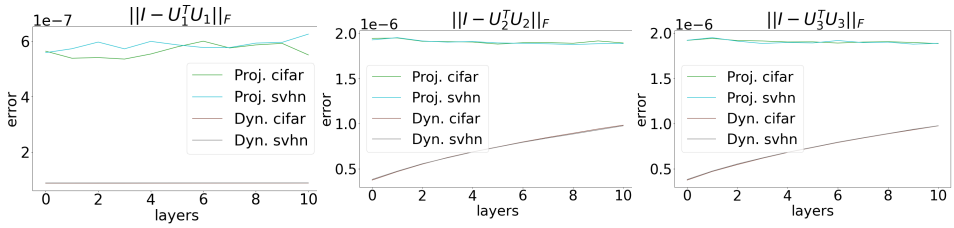
Figure 4.9: Convergence of the standard ResNet, ProjectionNet (Proj.) and DynamicNet (Dyn.) on CIFAR10 and SVHN.

	ResNet		ProjectionNet		DynamicNet	
	CIFAR10	SVHN	CIFAR10	SVHN	CIFAR10	SVHN
Training accuracy	99.46	83.87	100.0	99.80	57.93	31.46
Validation Accuracy	37.46	55.26	14.13	14.06	33.53	19.80
Time	2920s	3083s	377s	374s	1598s	1562s

Table 4.4: Table of maximum training and validation accuracy for the different networks on CIFAR10 and SVHN, time taken to run 30 epochs.

of size $U_1 \in \mathbb{R}^{3\times 3}$, $U_2 \in \mathbb{R}^{k\times m}$ and $U_3 \in \mathbb{R}^{k\times n}$.

Then we initialise the networks. As the input to ResNet and DynamicNet is the largest, this network also requires many more trainable parameters, we have weight matrices of size $W_i^{[l]} \in \mathbb{R}^{3\times m\times n}$. The ProjectionNet has only weight matrices of size $W_i^{[l]} \in \mathbb{R}^{32\times k}$ by the wrapping procedure. Then we are ready to train and compare the networks, and the convergence can be seen in Figure 4.9. It is clear from Figure 4.9 that the methods behave very differently. The DynamicNets have the slowest convergence, and the final accuracy is low for both data sets. The convergence of the training accuracy seem to be very similar for both ResNet and ProjectionNet. The validation accuracies, however, are very low for the ProjectionNets and decent for the ResNets.



(a) U_1 **(b)** U_2 **(c)** U_3

Figure 4.10: Orthogonality error for ProjectionNet and DynamicNet on CIFAR10 and SVHN for each layer of the trained network.

In Table 4.4, we see the maximum training and validation accuracies for the networks. The performance of DynamicNet on CIFAR is close to the performance of ResNet, with only a few percent difference. The same cannot be said for ProjectionNet. There is no doubt that the methods have been implemented purely for readability and correctness rather than speed, which again reduces the computational efficiency.

We also show numerical evidence of the orthogonality of U_1, U_2 and U_3 in both ProjectionNet and DynamicNet, see Figure 4.10(a), 4.10(b) and 4.10(c). From the orthogonality plots, the matrices U_1, U_2 and U_3 remain orthogonal throughout the network, as expected. There is not much variation in the orthogonality through the layers.

This difference could be one of the differences between the continuous model and the non-continuous.

4.5 Adversarial Robustness

Now we will continue with experiments on adversarial robustness, in particular the Fast Gradient Sign Method. The structure will follow the previous section, first looking at only low-rank cases, and then also orthogonal restrictions.

The attack procedure is as follows. We choose a list of perturbations in the data range $\epsilon \in [0, 1]$ to run for all experiments. For illustrations purposes, we choose to keep $\epsilon = 0$, as it represents the model performance on the test set. We expect that for larger perturbations of ϵ the more noticeable the noise is in the image, and also the more effective the attack will be in terms of degrading the model performance [45, 25]. For each value of $\epsilon > 0$ we run the test set through the network. If the network makes a correct prediction, the FGSM will attack. The method creates a perturbed adversary. We run

the perturbed example through the network and check if this example is adversarial. We keep track of the accuracy for each ϵ. This is done for each network, and therefore we attack each networks vulnerabilities.

4.5.1 Low-Rank

Recall that we are now investigating the standard ResNet, and compressing only input to the low-rank manifold, or restricting at every layer. Their performance can be found in Figure . We compare and attack the three methods. The results can be seen in Figure 4.11. On MNIST the restricted and compressed networks have the largest drop in accuracy for $\epsilon = 0.05$. The standard ResNet is not as badly affected as the other two methods for small ϵ. At $\epsilon = 0.01$ the restricted-100 and the ResNet-100 performs the best. Note how the restricted networks have flattened their drop in accuracy, and restricted-100 is now the best performing network of all. For $\epsilon > 0.1$ the restricted networks perform significantly better than compressed ResNet and ResNet. The results are similar for FashionMNNIST but the differences are not so prominent. Regarding the depths of the network, there might be a slight improvement using deeper networks against adversarial attacks. For CIFAR10 and SVHN the results are disappointing. The accuracy for these networks is low, but also the accuracy of all attacks is almost zero. Note, however, that restricted and ResNet on SVHN has slightly better accuracy for $\epsilon = 0.3$. This could be insignificant. We can investigate the generated perturbed examples, and we notice that they look like the generated examples the standard ResNet in Figures 4.13(a) (MNIST) and Figure 4.14(a) (FashionMNIST).

4.5.2 Low-Rank and Stiefel

We now turn our attention to ResNet, DynamicNet and ProjectionNets robustness towards the FGSM. However, as the results from the networks on CIFAR10 and SVHN was not sufficient, we will not run this test on tensors, as the methods has not found sufficient patterns for classification.

MNIST vs FashionMNIST The results of the FGS Attacks on MNIST and FashionMNIST can be seen in Figure 4.12. It is clear that the accuracy deteriorates rapidly for small perturbations to the image for all methods on both data sets. Notice, however, that both ProjectionNet and DynamicNet perform better for $\epsilon \geqslant 0.15$. In the case of ResNet, the accuracy drops to below 10%. This suggests that the attacks are effectively fooling the network into misclassification. This is not true for ProjectionNet and DynamicNet, which have accuracy around and above 10% for all networks on both data

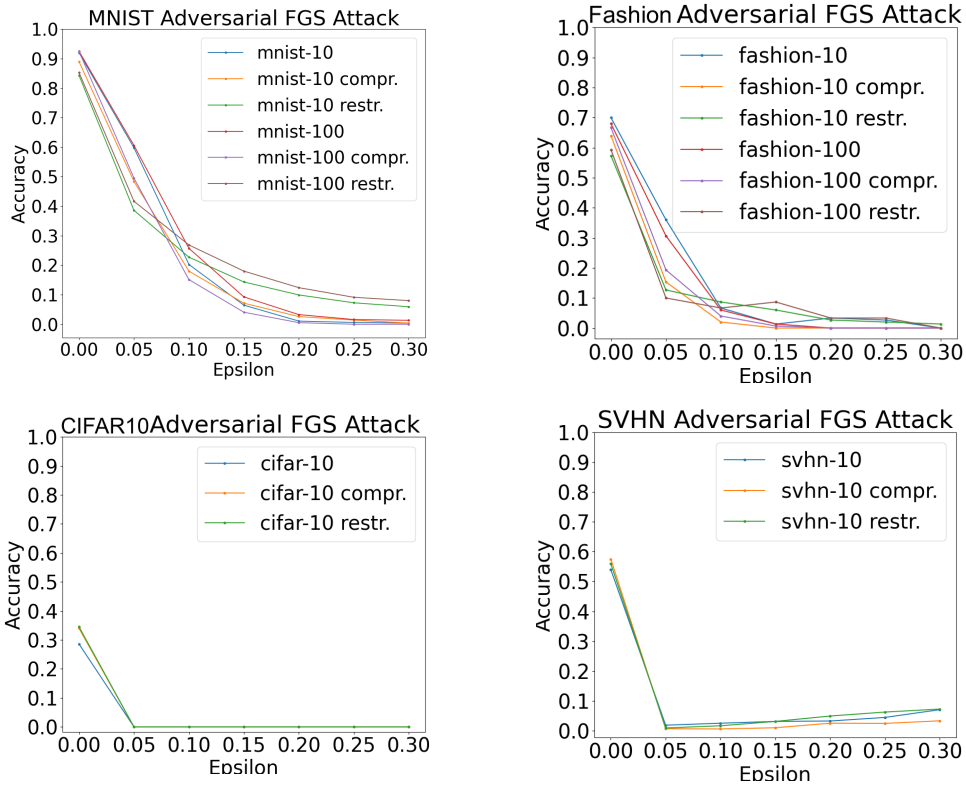


Figure 4.11: Accuracy of attempted Adversarial FGS Attacks for each value of epsilon.

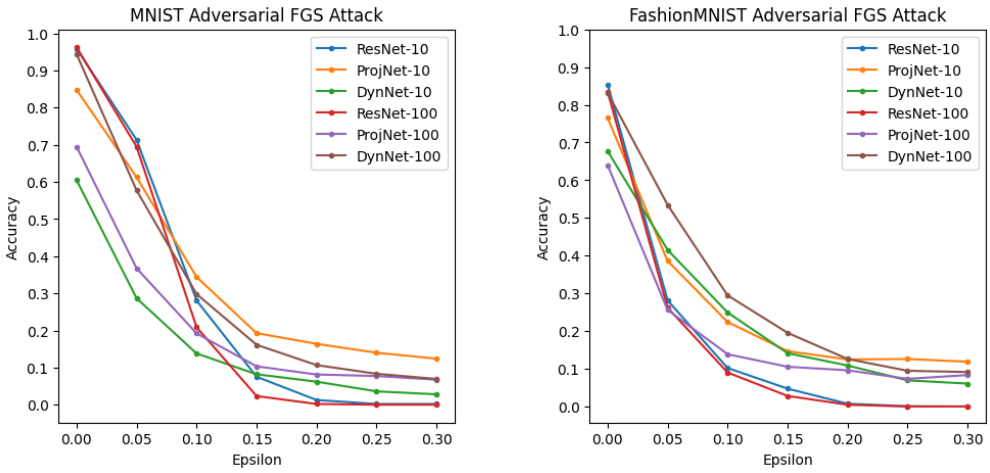
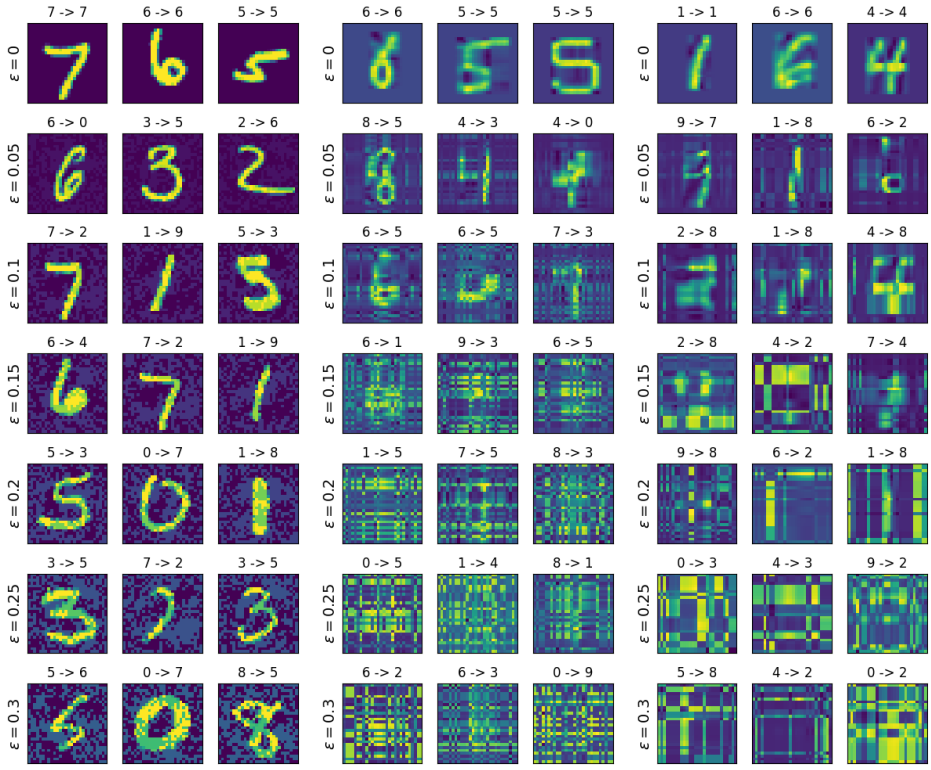


Figure 4.12: Accuracy of attempted Adversarial FGS Attacks for each value of epsilon.

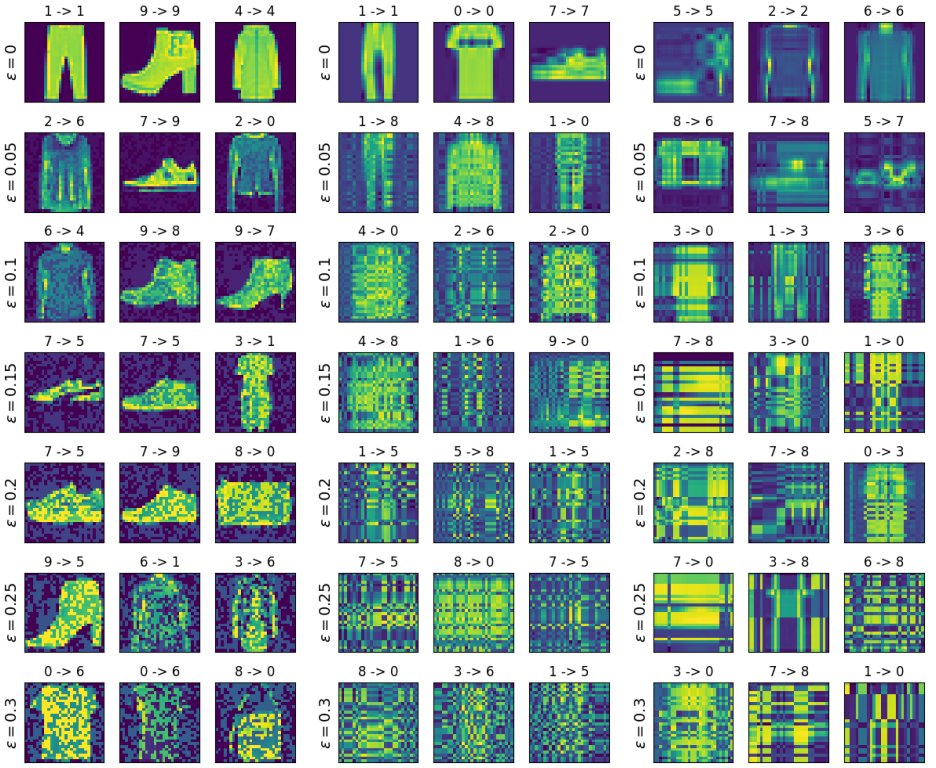
sets. ProjectionNet10 has the worst accuracy of all the methods, but is the most robust to the attacks at least in MNIST. The effects of the attacks do not seem to deteriorate the accuracy in a similar manner as in DynamicNet. For DynamicNet10, the accuracy drops far lower than the rest of the methods for small epsilon. There is also a slight difference between the deep and shallow networks. It seems that the deeper networks are more robust to adversarial attacks for all ϵ, this is visible in DynamicNet100. This is true in general, but not for the shallow ProjectionNet-10. These experiments on adversarial attacks should be performed a few times and averaged such that we avoid the random effect. This was not done in these experiments, as the procedure to run the attacks proved to be time consuming. It would also be valuable to investigate the effects on size of training data sets and number of epochs.

In Figure 4.13 we can see examples of generated MNIST adversaries by the different networks. The examples generated by deeper networks look very similar, if not identical. Investigating the adversaries generated by the ResNet, it is clear that FGSM is adding what we perceive as noise to the image to successfully fool the network into misclassification. For each value of ϵ the image becomes noisier. These generated examples stand in clear contrast to the examples generated by the Projection- and DynamicNet. The adversaries generated by these two methods are for $\epsilon > 0$ hard for a human to classify. These images have been added more noise then the corresponding



(a) ResNet **(b)** ProjectionNet **(c)** DynamicNet

Figure 4.13: Examples of generated adversaries for each value of ϵ for MNIST. The numbers above each image represent the original class and the predicted class.



(a) ResNet **(b)** ProjectionNet **(c)** DynamicNet

Figure 4.14: Examples of generated adversaries for each value of ϵ for FashionMNIST. The numbers above each image represent the original class and the predicted class.

FGSM on ResNet. It is clear that for larger values of ϵ the FGSM is struggling to add noise which does not completely modify the perturbed image making it unrecognisable to a human. These generated adversaries are not fooling the ProjectionNet and DynamicNet as with ResNet, as we can see from Figure 4.12. However, it is not clear from these examples if this is a pure benefit. There are still many misclassifications for $\epsilon < 0.15$.

We can see similar results for the generated adversarial examples by the networks on FashionMNIST, see Figure 4.14. By looking at both Figures 4.13 and 4.14, it is clear that there is a certain structure to the "noise" added by ProjectionNet and DynamicNet to the adversarial examples. This structure can be seen as patches of smaller and lager squares in to the image. The structured squares added by DynamicNet is larger than the structure

added by ProjectionNet. This difference is significant, yet why this difference in structure occurs, and the magnitude of it is still unknown. Investigating this structure, if it is high- or low-rank, orthogonality properties as well as the weight and bias would give us an idea of what is happening internally in the network.

Chapter 5

Discussion and Future Work

Throw up into your typewriter every morning.
Clean up every noon.
– Raymond Chandler

It goes without saying that many presented aspects deserve to be discussed. For readability, we divide the discussion into three; background, results, and future work. In the background, we will reflect on the methods used and the pros and cons. In the results, we will discuss the results presented in the previous chapter, focusing on the integration techniques, adversarial robustness and regularisation. Then we discuss points for further work.

5.1 Background

We presented much motivation behind choosing the methods we have focused on: the SVD, Polar Projection, Dynamic Low-Rank Approximation and Dynamic Tensor Approximation. There is, at the moment, no deep mathematical understanding of the benefits of preservation of low-rank structure or orthogonality. Some papers, such as [10], inspire us to try to preserve structures in the development of neural networks. By structure-preserving methods, we mean that the evolution of the is performed on the manifold. In our case, the low-rank or the Stiefel manifold. The benefits of a particular invariant might be problem-specific. Even if we know an invariant is essential to maintain, how do we enforce it? We can insist on low-rank U and V by truncating them, leaving out the orthogonality constraint. Or focus on the different integration techniques presented in ProjectionNet and

DynamicNet, illustrating the many ways to achieve the desired structure. Each method with its implications, both at the network design level and for robustness. Understanding simpler models like this linear network may be a good starting point for gaining theoretical insights.

We argued in Chapter 1 that there might be reduced order benefits if we can establish a connection between data sets and the lower-dimensional manifold hypothesis. If we assume the existence of a lower-dimensional manifold structure in the data, we have seen that it is not easy to determine whether the approximation is a good enough representation. In the previous chapter, we wanted to find a truncation k for the SVD and Tucker decomposition. We quickly noticed that we had to make a choice of truncation k, which was not rigorously justified. Which truncation to choose was primarily due to the error we were willing to introduce into the images and subjective differences between the truncated images. It is clear that finding this representation of the underlying manifold, whether a coordinate change or a low-rank approximation is difficult.

There is a possibility that the more restrictions are put on our network, we increase the likelihood of deteriorating the speed of convergence and the chance of recovering global minima. These restrictions are possibly changing the optimisation landscape. It might be that structure preserving methods are too strict when we require orthogonality and low-rank. However, when it comes to orthogonality, many papers have seen an increase in the convergence rate when keeping the weight orthogonal [61, 31, 42, 1, 3]. We also know that the eigenvalues of orthogonal matrices are on the unit circle [26, 50]. Thus, imposing orthogonality we can also aid in stability of the methods, as we know that stable networks have negative real eigenvalues. Also, low-rank projections are costly. But, we will tolerate an increase in computation time if the methods have obvious benefits, like robustness. We suspect that many standard image data sets can be well approximated by their projection on low-rank spaces in neural networks, as demonstrated in experiments.

The SVD has proven advantageous in many fields and is a safe first choice. Another interesting approach would be to investigate the effects of a Fourier Transform or Fourier Expansion on the neural network's data sets. The Fast Fourier Transform is a popular tool in signal processing and is well understood.The generalised Fourier series form a complete orthogonal system. It might be possible to combine the network with a Fourier transform to only extract extract the essential signals in the image, which serves as an orthogonal basis. Combining the SVD with a Neural Network, and in par-

ticular, the residual neural network, we can learn and unfold the manifold enabling the classification of its parts. However, in manifold learning, the SVD and the related PCA are not popular choices, as linear transformations have proven to not be capable of learning non-linear structures. Therefore, it would be interesting to investigate methods such as Local Linear Embedding and Isomap, as seen in [9, 15, 74].

Furthermore, it might not be necessary to project at every layer. Many modern-day neural networks use a combination of linear and convolutional layers and encoders and decoders as blocks in neural networks. Networks such as encoders and decoders can be used to enhance the performance of linear dimension reduction techniques [49]. It might be possible to benefit structure preservation by combining blocks of the ResNet where preserve the invariants.

Dupont [16] suggested feature space augmentation to overcome topological challenges such as nested objects in the data sets. Apart from seeing how feature space augmentation affects the networks' performance, it would be interesting to investigate the benefits. Can we find the number of nested structures in our image manifold to determine a lower bound for the number of additional dimensions we need to modify the feature space with? Is there a relation between nested structures, ranks, holes, etc.?

Solving differential equations on manifolds is cheaper, in theory, than computing the SVD at every time step. Koch and Lubich argue that we are multiplying matrices with fewer columns in the case of Dynamic Low-Rank [52]. However, it is clear from our results that the DynamicNets are many times slower than ResNet and ProjectionNet, as seen in Table 4.2 and 4.3. This makes it hard to believe that it is possible to gain speed past standard ResNet for as long as we have an equal amount of trainable parameters and the need to reassemble the image at every layer. However, when running on CIFAR and SVHN we noticed that DynamicNet was a lot faster than ResNet, see Table 4.4. This result is surprising. One explanation could be that the methods have converged after few epochs. Therefore the cost during backpropagation is minimal, as there are no further updates to the parameters other than small changes as seen in the plot. Therefore, it is less time consuming. It could still be worthwhile to investigate a more efficient way to represent, multiply and store the network's matrices, such as in ProjectionNet. The difference between the integration step in DynamicNet and ProjectionNet is interesting from a mathematical point of view. The Dynamic Low-Rank Approximation method might respect the flow of the neural network, as this method yields continuous smooth evolution on

the manifold. If this is the desired behaviour is another question, but the significant difference between the methods is worth pondering.

One of the main differences between ProjectionNet and DynamicNet is the unique formulation of the equations 2.11 for DynamicNet. The Dynamic Low-Rank approximation gives an optimal representation of the generated vector field which is generated by the Network. The optimally is in the Frobenius norm of the difference between the two vector fields. Whether this is true for ProjectionNet is not known, and what the consequences are for the method. We can also think of uniqueness and smoothness as structures that we might be keen on preserving, with their own properties. One thing worth noting is that the initial condition is not unique, see PyTorch's' documentation[1] . This means that a given image might have more than one SVD and Tucker Decomposition, as stated in Chapter 2. Before training starts the data sets are prepared, and thus images are truncated or factorised. This is done every time we run a network, or other algorithms that require access to the data. Consequently, with its different decompositions, this image might have multiple trajectories in the network depending on its decomposition.

When it comes to the implementation, we have experienced difficulties when back propagating the inverse, especially the Cayley transform in deeper networks. Our solution was to perform the exponential maps instead. The same care should be taken with SVD of the images, in the case of repeating singular values. There is a possibility that these operations are not differentiable. In [52, 29], Koch and Lubich investigated singular values and discontinuities. Singular values and eigenvalues are essential for the stability of the trained vector field. Thus, studying the evolution of the singular values for DynamicNet and ProjectionNet could give further insights into the differences between these methods.

5.2 Results

We have successfully trained three different networks on a compressed, truncated, and even factorised form of the data, see Figure 4.4, Figure 4.6 and Table 4.1. This can be viewed as an indication of a lower-dimensional structure in the data. However, we remain uncertain to what we should give credit; the data or the method. It might be that the network is so good at finding hidden structures that most of the pixels are superfluous, even if there is not a lower-dimensional structure in the data.

[1] https://pytorch.org/docs/stable/generated/torch.linalg.svd.html

MNIST and FashionMNIST The convergence of accuracy plot of MNIST and FashionMNIST, Figures 4.7(a) and A.5(a) show the methods converging nicely towards perfect training accuracy. In this plot the ProjectionNet has a slow convergence, and the validation accuracy settles on a far lower accuracy than the other methods. From Tables 4.2 and 4.3, it is intriguing to see that DynamicNet performs as well as ResNet on the validation data on both data sets and for both depths. This is not true for ProjectionNet, and the validation accuracy has deteriorated by $5-10\%$. This significant loss in accuracy is not straightforward to explain. The most apparent reason could be the number of trainable parameters. ProjectionNet has only layers of size $28 \times k$, whereas ResNet and DynamicNet have full-size layers 28×28. This is not the whole truth, as the validation accuracy is poor without much increase in the deeper network. This might still be the explanation if there is a different contribution between width parameters and depth parameters. We can suspect this to be true if we revisit the work by Dupont [16], where he illustrates the benefit of augmenting the width. This is also very common in convolutional layers, where we augment the number of channels. There are also some minor differences due to random initialisation. To establish the results, we need to verify for a few more data sets, do a few runs and average the results. However, this is just a suspicion, and we need to conduct experiments on this to be sure. We will return to this after investigating the results on CIFAR and SVHN.

The plots of orthogonality and ranks see Figure 4.7, 4.8(a), 4.8(b)and Figure A.5(b), A.5(c), A.5(d), are the only means of verifying that the data is transformed within the desired manifold. The error in orthogonality is typically between $1e-6$ and $1e-7$. As the orthogonality is not preserved to machine accuracy, this could be the reason for the slight performance to drop of DynamicNet compared to ResNet on MNIST and FashionMNIST, as seen in Table 4.2 and Table 4.3. There might be benefits of reducing this error.

Another big difference between ProjectionNet and DynamicNet is the reassembling of the image components at every layer. In DynamicNet, the factors are propagated in the manifold and then reassembled. In ProjectionNet, the image is only assembled at the very last stage of the network. This might be the crucial difference, both performance-wise and computationally. We could establish a hybrid method. Propagating the matrices separately for a few layers before combining them to maintain accuracy. Now that we know that there are certain robustness benefits, this approach is interesting to explore.

CIFAR10 and SVHN The accuracy plot of CIFAR10 and SVHN, Figure 4.9, shows all methods are having trouble converging to the desired accuracy. ResNets CIFAR10 and SVHN training accuracy converge slowly but steadily, but their validation accuracy has settled much lower. We can also see the ProjectionNets on CIFAR10 and SVHN are gradually converging towards an acceptable training accuracy. However, the validation accuracy has settled around 10%. For DynamicNet, both training- and validation accuracy has converged to a disappointingly low performance. Taking a quick look at the orthogonality plots, Figures 4.10(a), 4.10(b) and 4.10(c) the factors satisfy the orthogonality constraint, as expected.

Increasing the number of epochs might increase the likelihood of escaping from saddle points, especially when adding momentum to the optimiser. This is perhaps more relevant in the case of MNIST and FashionMNIST. This effect can give a few points the accuracy rate of MNIST and FashionMNIST, but we will not see methods becoming miraculously better in the case of CIFAR10 and SVHN. There might be structures in the data sets which are too complex for the linear layered networks to determine. It is clear that DynamicNet performs as well as ResNet in the case of MNIST and FashionMNIST. Therefore, we should expect the same behaviour here. This is not the case. There are a few potential reasons for the bad performance. Even if the orthogonal matrices evolve on the Stiefel manifold, we have a considerable-sized component we do not control: the core tensor $\boldsymbol{\mathcal{S}}$. This could potentially be the cause of the bad performance. A potential fix for this will be discussed in the next section. Lastly, we also expected the ProjectionNet to be performing better. However, in MNIST and FashionMNIST, the validation accuracy for ProjectionNet was significantly lower than the other two methods. Also, recall that we had to construct a padding around the Tucker decomposition in order to get the network to handle the different sizes of matrices and tensors, (3.18) and (3.3.1). This padding is potentially deteriorating the learning of the orthogonal matrix U_1. PyTorch is also flattening each channel of the core tensor into a vector. The vector field defining the network is essentially handling each channel of the core tensor $\boldsymbol{\mathcal{S}}$ separately. Therefore, we could expect the method to have a similar result here.

We are confident that changing the implementation of the tensor networks can lead us to better performance. This is also due to the results on the compressed and restricted data, see Figure 4.6. Where it seems that both restricted and compressed are performing as well as ResNet.

Another big difference between ProjectionNet and DynamicNet is the re-

assembling of the image components at every layer. In DynamicNet, the factors are propagated in the manifold and then reassembled. In ProjectionNet, the image is only assembled at the very last stage of the network. This might be the crucial difference, both performance-wise and computationally. Could we establish a hybrid method, propagating the matrices separately for a few layers before combining them to maintain accuracy?

In the experiments, the truncation was chosen so that the error between the original and truncated images would be roughly the same in all data sets. We noticed that in the case of CIFAR10 and SVHN the ProjectionNet is not performing nearly as well. So as the error in all truncation was roughly the same, we can suspect that it is not a lack of information that makes the network perform poorly. In the ProjectionNets, the truncation of the image also indicates how many trainable parameters the network has, as already mentioned. In a paper by Bubeck [7], one shows a measure of the number of parameters needed in neural networks. The author also theoretically explain the need for over-parametrisation, which is very common in deep learning models. It is tempting to adopt their point of view and see if there is a lower bound on how much we can truncate the input for ProjectionNet based on the work by [7]. If so, the next question becomes; if we need more parameters per layer than we provided in our ProjectionNet, what is the difference between choosing a larger k and thus including more information back into the image, or just padding with zeroes using a feature space augmentation? How much "information" do we need to train a network?

Regularisation and Adversarial Robustness Recall the the plots of only low-rank restrictions, Figure 4.11, to both low-rank and orthogonality restrictions Figure 4.12. We see that on MNIST in 4.11, the compression and restriction methods work well for larger epsilon. However, note that their accuracy deteriorates fast for smaller epsilon. This is also true for FashionMNIST. Shifting our focus towards Figure 4.12 we see that the Dynamic and ProjectionNet performs better than restricted and compressed for smaller ϵ. The relative drop in performance for increasing ϵ is smaller than for ResNet and restricted and compression. All of the methods conserving low-rank are more robust to larger ϵ. Thus, we see that the latter are more robust to adversarial attacks. The most obvious explanation is the orthogonality restriction combined with the low-rank.

One benefit of having constant width networks, is that we can keep track of the visual changes the networks do to each image. It could be intuitively pleasing to investigate the visual changes, especially if it correlates with what

features the network enhances or not. However, one can expect the results not to be so intriguing and that the differences between outputs at every layer are relatively small. After all, the step size, and thus the modification added to the identity map, is very small for each layer. However, there might indeed be a particular structure to this modification, even if this is not detectable to the naked eye, as in 4.13(a) and 4.14(a). This structure difference might be the very thing that makes the Adversarial FGS Attacks of ProjectionNet and DynamicNet very different from ResNet. Suppose we perform SVD at every layer, and the output of each layer is a very similar-looking image. In that case, we are, in some sense, training the network on a variety of these very similar images. This idea works in the case of standard ResNet and Projection- and DynamicNet. For each layer in the network, the modifications done by the layer are seemingly random. But the further we propagate the network through the layers, the more random noise is added to the image. Therefore, the deep networks are more robust, as they have seen more of this random noise in the training process. For Projection- and DynamicNet, this added structure through the layers create completely different weak points that the Adversarial FGSM can attack. For this reason, the perturbed images look very different. For the FGSM to maximise the loss function and create an effective adversarial attack, we see that the FGSM must perturb the image to such an extent that it becomes unrecognisable to the human eye. The attack is, in some sense, not working. The perturbation to the image is not subtle, and the network is not fooled into misclassification. Unfortunately, we do not have an explanation for this. Fewer parameters will also make the options for weak nodes to attack in the network smaller. Now the question becomes; which structure is least prone to attacks? Before we can answer this, there are many questions we need to answer regarding what we define as adversarial [80, 77]. Which types of attacks are we expecting, and how should we deal with these attacks. It is intriguing to investigate these generated examples' rank and orthogonality. If this is true, is it possible to restrict the function space in which the network is trained, which again limits the possible adversarial examples we need to handle?

Recall the generated adversarial examples by the networks, as seen in Figures 4.13 and 4.14. In ResNets we do not have to change the input much to make the network misclassify. But in ProjectionNet and DynamicNet, the perturbed images are remarkably different from the input. We can define successfully adversarial attacks as fooling the network into misclassification, but the generated example is noisy to a human. If this is the definition we choose, then in the case of DynamicNet and ProjectionNet, the attacks fail

for larger ϵ.

The results for the adversarial attacks on SVHN and CIFAR are not sufficient, see Figure 4.11, and we need to boost the validation accuracy before we can continue the discussion for tensors.

There seems to be a connection between the smoothness of networks, Lipschitz regularisation, stability of ODES and Adversarial Attacks. Adversarial Attacks are closely related to the stability of ODES; a small change to input should not result in large changes to the output. The way to measure Adversarial Attacks via FGSM is just by determining examples where the network, or the function, has particularly weak stability. Is there a connection between Lipchitz regularisation of the model parameters, weight regularisation and regularisation of the output produced at each model layer? Could it be that Low-Rank projections and/or orthogonality constraints induce a Lipschitz regularity in the network? In that case, we can connect our results to the results of weight regularisation and the convergence of Neural ODES. Recall that [75] used a regularisation term to show convergence towards the continuous solution. If there is a relation between regularisation and low-rank projections, then the results still hold.

We might suspect that the loss surface of the DynamicNet and ProjectionNets are flatter if the result of the restriction to the manifold indeed has a regularising effect, as the convergence is slower and we do not retrieve the same local minima. In that case, we also might expect a difference between the ResNet and the Projection- and DynamicNet's loss function. This can also be suspected by looking at the slower convergence of these methods. Different optimisation landscapes provide different convergence rates. Also, the same local minima do not exist. We have seen that low-rank input into a ResNet finds the same local minimum as a high-rank input. However, when we restrict the evolution of the vector field, we are also altering the optimisation landscape, resulting in a more regularised loss function with different local minima.

In [57] the author investigates what happens to the singular value, in particular the relative change in the singular values, when adversarial perturbations are added. They find that the smaller singular values gain the most in relative change. And the adversarial perturbations tend to be full rank. They also provide an explanation of why compression (SVD) reverses small adversarial perturbations. This paper also argues that SVD compression aids in adversarial robustness but is not enough to completely prevent it [17]. When truncating the images, we are, in some sense, removing noise and

high frequencies. However, this is not in accordance with our findings for linear networks. In our case, both truncating the input and restricting only to the low-rank manifold do not seem to improve the robustness to FGS Attacks. The deep ResNet-100 restricted on MNIST is more robust, but this is not true for shallow restricted networks and not for FashionMNIST.

As we have seen, the methods preserving orthogonality are more robust to Fast Gradient Sign Attacks than ResNet. In MNIST ProjectionNet-10 is clearly the best method, whereas in FashionMNIST DynamicNet-100 outperforms the others, even for small perturbations. An important remark is to notice the performance of DynamicNet-10 on MNIST, see Figure 4.12. The accuracy of the method at $\epsilon = 0$ is worse than the accuracy after training, as seen in Table 4.2. This is very unexpected, and does not seem to be the case for the deeper DynamicNet. This is sightly concerning, and we dont have a good explaination for this.

As all of these networks have linear layers, they are naturally more prone to adversarial attacks, as described in [25]. They demonstrate that linear networks are more prone to adversarial attacks if the input has sufficient dimensionality. ProjectionNet has fewer parameters to attack, making it less prone to FGS attacks. Furthermore, it might seem that the deeper networks are more robust to the attacks. The deep networks contain more nonlinearities. Note however, that the shallow ProjectionNet outperforms other networks on MNIST, so there might be other explanations for this behaviour.

5.3 Future work:

When we input the images into PyTorch's layers, the built-in method from PyTorch is to flatten the image. This flattening has unknown consequences, but intuitively this operation disregards the manifold structure treating matrices and tensors simply as large vectors. This is in particular nearness between points. However, we can also establish that we humans do not learn like neural networks. The networks have their way of finding structures that enable classification, which is entirely different from humans. PyTorch does allow for constructing self-made layers. This means we could build a matrix layer and a tensor layer, which could potentially conserve these topological structures. In this way, we can also avoid the padding introduced in the Tucker decomposition. We can construct a stack of layers for the core tensor $\boldsymbol{\mathcal{S}}$ or the matrix Σ and a stack of layers specialising in the orthogonal matrices. A way to combine them at the end is to have a linear classifier that combines all the trained outputs from each parallel network and recon-

structs the image. This might allow for more specialised networks, and also potentially reduce some parameters. The idea behind training networks on the decompositions is interesting, and there are still many directions to go.

ProjectionNet has some interesting abilities. It would be interesting to see how the networks change behaviour by increasing the width. If it can be augmented by simply padding the width with zeros, or if we must increase the width by increasing the truncation k. Investigating this problem could lead us to more certainty about what is required from input data and what is required from the networks.

It would be interesting to see the results with adaptive step size, the vector $h \in \mathbb{R}^L$, where we train a step-size for each layer. Construct a separate network for S and Sigma. This will reduce the wrapping in tucker decomposition and hopefully provide better and more correct control of S and Sigma.

The DynamicNet can also be inspired by ProjectionNet. Instead of reassembling the image at every layer, we can input the factors to the vector field as done in ProjectionNet (3.16). This might also mean that we do not need the Dynamic Low-rank formulation and instead take the Lie-Group approach on the ProjectionNet.

There seems to be potential regarding stability and adversarial robustness by preserving orthogonality in this specific way, as seen in our experiments. Studying the generated examples, the Lipschitz constant in the network, and the singular values could be an interesting direction.

Chapter 6

Conclusion

After climbing a great hill, one only finds
that there are many more hills to climb.
– Nelson Mandela

In this thesis, we have considered deep learning as the continuous optimal control problem. Two approaches to numerical geometric integration in neural networks have been studied. These methods aim to preserve the rank and orthogonality of the feature spaces through the network.

We started with the observation that truncating the image input of the data sets did not result in a deterioration of performance in standard residual neural networks. We hypothesised that the images could be well represented on a lower-dimensional manifold due to the low-dimensional Manifold Hypothesis. Therefore, we started experimenting with methods that could take advantage of the Singular Value Decomposition as coordinates on this low-rank manifold.

We chose to conserve both orthogonality and low-rank of the SVD through geometric numerical integration. Two integration approaches were considered, thus resulting in two different methods. One local-coordinates approach and one Euclidean approach combined with a simple projection method. The latter could also be an order-reduction method, aiming at reducing the cost of the learning problem as well.

We found that the developed methods indeed work as well, or almost as well, as the original ResNet formulation on matrices. The methods did not

show good results on tensors, but we are optimistic that this is mostly due to the choices made during the implementation.

The methods show promising results regarding robustness to adversarial attacks. The experiments show in particular that the manifold methods are not fooled for larger perturbations to the input data. They also show that some of the methods are more robust than ResNet for smaller perturbations as well.

These results are intriguing and could be a step in the right direction, not only to develop a better understanding of structure preserving methods within deep neural networks but also to increase robustness against adversarial attacks.

Goodfellow et.al write in their paper, [25], on adversarial attacks: "These results suggest that classifiers based on modern machine learning techniques, [..], are not learning the true underlying concepts that determine the correct output label. Instead, these algorithms have built a Potemkin village that works well on naturally occurring data, but is exposed as a fake when one visits points in space that do not have high probability in the data distribution [25]". Hopefully, by studying and learning the data manifold, we are not building Goodfellow's Potemkin village. In the process, we are also, dealing with valuable feature space maps that have a certain structure. This structure can be utilised to ensure properties in the neural networks, such as stability and convergence.

Bibliography

[1] Traian E. Abrudan, Jan Eriksson and Visa Koivunen. 'Steepest Descent Algorithms for Optimization Under Unitary Matrix Constraint'. In: *IEEE Transactions on Signal Processing* 56.3 (2008), pp. 1134–1147. DOI: 10.1109/TSP.2007.908999.

[2] P.-A. Absil, R. Mahony and Rodolphe Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, 2009. ISBN: 9781400830244. DOI: doi:10.1515/9781400830244. URL: https://doi.org/10.1515/9781400830244.

[3] Nitin Bansal, Xiaohan Chen and Zhangyang Wang. *Can We Gain More from Orthogonality Regularizations in Training Deep CNNs?* 2018. DOI: 10.48550/ARXIV.1810.09102. URL: https://arxiv.org/abs/1810.09102.

[4] Ronen Basri and David Jacobs. *Efficient Representation of Low-Dimensional Manifolds using Deep Networks*. 2016. DOI: 10.48550/ARXIV.1602.04723. URL: https://arxiv.org/abs/1602.04723.

[5] Y. Bengio, P. Simard and P. Frasconi. 'Learning long-term dependencies with gradient descent is difficult'. In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[6] Martin Benning, Elena Celledoni, Matthias J. Ehrhardt, Brynjulf Owren and Carola-Bibiane Schönlieb. *Deep learning as optimal control problems: Models and numerical methods*. 2019.

[7] Sébastien Bubeck and Mark Sellke. *A Universal Law of Robustness via Isoperimetry*. 2021. DOI: 10.48550/ARXIV.2105.12806. URL: https://arxiv.org/abs/2105.12806.

[8] Gunnar Carlsson. 'Topology and Data'. In: *Bulletin of The American Mathematical Society - BULL AMER MATH SOC* 46 (Apr. 2009), pp. 255–308. DOI: 10.1090/S0273-0979-09-01249-X.

[9] Lawrence Cayton. 'Algorithms for manifold learning'. In: 2005.

[10] Elena Celledoni, Matthias J. Ehrhardt, Christian Etmann, Robert I McLachlan, Brynjulf Owren, Carola-Bibiane Schönlieb and Ferdia Sherry. *Structure preserving deep learning*. 2020. arXiv: 2006.03364 [cs.LG].

[11] Lieven De Lathauwer, Bart De Moor and Joos Vandewalle. 'A Multilinear Singular Value Decomposition'. In: *SIAM Journal on Matrix Analysis and Applications* 21.4 (2000), pp. 1253–1278. DOI: 10.1137/S0895479896305696. eprint: https://doi.org/10.1137/S0895479896305696. URL: https://doi.org/10.1137/S0895479896305696.

[12] DeepAI. *Manifold hypothesis*. May 2019. URL: https://deepai.org/machine-learning-glossary-and-terms/manifold-hypothesis.

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

[14] Luca Dieci and Timo Eirola. 'On Smooth Decompositions of Matrices'. In: *SIAM J. Matrix Anal. Appl.* 20 (1999), pp. 800–819.

[15] R.O. Duda, P.E. Hart, P.E. Hart, P.E. Hart, D.G. Stork, Ebook Library and John Wiley & Sons. *Pattern Classification*. A Wiley-interscience publication poeng 1. Wiley, 2001. ISBN: 9780471056690. URL: https://books.google.no/books?id=YoxQAAAAMAAJ.

[16] Emilien Dupont, Arnaud Doucet and Yee Whye Teh. *Augmented Neural ODEs*. 2019. DOI: 10.48550/ARXIV.1904.01681. URL: https://arxiv.org/abs/1904.01681.

[17] Gintare Karolina Dziugaite, Zoubin Ghahramani and Daniel M. Roy. *A study of the effect of JPG compression on adversarial images*. 2016. DOI: 10.48550/ARXIV.1608.00853. URL: https://arxiv.org/abs/1608.00853.

[18] Carl Eckart and Gale Young. 'The approximation of one matrix by another of lower rank'. In: *Psychometrika* 1.3 (1936), pp. 211–218.

[19] Charles Fefferman, Sanjoy Mitter and Hariharan Narayanan. *Testing the Manifold Hypothesis*. 2013. DOI: 10.48550/ARXIV.1310.0425. URL: https://arxiv.org/abs/1310.0425.

[20] E. Fiesler. 'Neural Network Classification and Formalization'. In: *Computer Standards & Interfaces* 16 (1994), pp. 231–239.

[21] Elisa Giesecke and Axel Kröner. *Classification with Runge-Kutta networks and feature space augmentation*. 2021. arXiv: 2104.02369 [cs.LG].

[22] Xavier Glorot and Yoshua Bengio. 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[23] Xavier Glorot, Antoine Bordes and Yoshua Bengio. 'Deep Sparse Rectifier Neural Networks'. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[24] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013. ISBN: 9781421407944. URL: https://books.google.no/books?id=X5YfsuCWpxMC.

[25] Ian J. Goodfellow, Jonathon Shlens and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2014. DOI: 10.48550/ARXIV.1412.6572. URL: https://arxiv.org/abs/1412.6572.

[26] Eldad Haber and Lars Ruthotto. 'Stable architectures for deep neural networks'. In: *Inverse Problems* 34.1 (Dec. 2017), p. 014004. ISSN: 1361-6420. DOI: 10.1088/1361-6420/aa9a90. URL: http://dx.doi.org/10.1088/1361-6420/aa9a90.

[27] Eldad Haber and Lars Ruthotto. 'Stable architectures for deep neural networks'. In: *Inverse Problems* 34.1 (Dec. 2017), p. 014004. DOI: 10.1088/1361-6420/aa9a90. URL: https://doi.org/10.1088%5C%2F1361-6420%5C%2Faa9a90.

[28] Eldad Haber, Lars Ruthotto, Elliot Holtham and Seong-Hwan Jun. *Learning across scales - A multiscale method for Convolution Neural Networks*. 2017. arXiv: 1703.02009 [cs.NE].

[29] Ernst Hairer, Christian Lubich and Gerhard Wanner. *Geometric numerical integration. Structure-preserving algorithms for ordinary differential equations. 2nd ed.* Vol. 31. Jan. 2006. ISBN: 3-540-30663-3. DOI: 10.1007/3-540-30666-8.

[30] Boris Hanin and David Rolnick. 'How to Start Training: The Effect of Initialization and Architecture'. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/d81f9c1be2e08964bf9f24b15f0e4900-Paper.pdf.

[31] Mehrtash Harandi and Basura Fernando. *Generalized BackPropagation, Étude De Cas: Orthogonality*. 2016. DOI: 10.48550/ARXIV.1611.05927. URL: https://arxiv.org/abs/1611.05927.

[32] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke and Travis E. Oliphant. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[33] Johan Håstad. 'Tensor rank is NP-complete'. In: *Journal of Algorithms* 11.4 (1990), pp. 644–654. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(90)90014-6. URL: https://www.sciencedirect.com/science/article/pii/0196677490900146.

[34] Juncai He, Richard Tsai and Rachel Ward. *Side-effects of Learning from Low Dimensional Data Embedded in an Euclidean Space*. 2022. DOI: 10.48550/ARXIV.2203.00614. URL: https://arxiv.org/abs/2203.00614.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV].

[37] Catherine F. Higham and Desmond J. Higham. *Deep Learning: An Introduction for Applied Mathematicians*. 2018. arXiv: 1801.05894 [math.HO].

[38] Desmond J. Higham. 'Time-stepping and preserving orthonormality'. In: *BIT Numerical Mathematics* 37 (Mar. 1997), pp. 24–36. URL: https://doi.org/10.1007/BF02510170.

[39] Nicholas J. Higham. 'Computing the Polar Decomposition—with Applications'. In: *SIAM Journal on Scientific and Statistical Computing* 7.4 (1986), pp. 1160–1174. DOI: 10.1137/0907079. eprint: https://doi.org/10.1137/0907079. URL: https://doi.org/10.1137/0907079.

[40] Nicholas John Higham. 'MATRIX NEARNESS PROBLEMS AND APPLICATIONS'. In: 1989.

[41] R.A. Horn and C.R. Johnson. *Matrix Analysis*. Cambridge University Press, 2012. ISBN: 9781139788885. URL: https://books.google.no/books?id=O7sgAwAAQBAJ.

[42] Lei Huang, Xianglong Liu, Bo Lang, Adams Wei Yu, Yongliang Wang and Bo Li. *Orthogonal Weight Normalization: Solution to Optimization over Multiple Dependent Stiefel Manifolds in Deep Neural Networks*. 2017. DOI: 10.48550/ARXIV.1709.06079. URL: https://arxiv.org/abs/1709.06079.

[43] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[44] Goodfellow Ian, Bengio Yoshua and Courville Aaron. *Deep Learning*. Adaptive Computation and Machine Learning. The MIT Press, 2016. ISBN: 9780262035613. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2565107&site=ehost-live.

[45] Nathan Inkawhich. *Adversarial Example Generation*. Aug. 2018. URL: https://pytorch.org/tutorials/beginner/fgsm_tutorial.html#adversarial-example-generation.

[46] Arieh Iserles, Hans Munthe-Kaas, Syvert Norsett and Antonella Zanna. 'Lie Group Methods'. In: *Acta Numerica* 9 (Jan. 2000), pp. 215–. DOI: 10.1017/S0962492900002154.

[47] Drahoslava Janovska and Kunio Tanabe. 'An algorithm for computing the Analytic Singular Value Decomposition'. In: (Nov. 2008).

[48] William Johnson and Joram Lindenstrauss. 'Extensions of Lipschitz maps into a Hilbert space'. In: *Contemporary Mathematics* 26 (Jan. 1984), pp. 189–206. DOI: 10.1090/conm/026/737400.

[49] N. Kambhatla and T.K. Leen. 'Fast nonlinear dimension reduction'. In: *IEEE International Conference on Neural Networks*. Vol. 3. 1993, pp. 1213–1218. DOI: 10.1109/ICNN.1993.298730.

[50] Qiyu Kang, Yang Song, Qinxu Ding and Wee Peng Tay. *Stable Neural ODE with Lyapunov-Stable Equilibrium Points for Defending Against Adversarial Attacks*. 2021. DOI: 10.48550/ARXIV.2110.12976. URL: https://arxiv.org/abs/2110.12976.

[51] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[52] Othmar Koch and Christian Lubich. 'Dynamical Low-Rank Approximation'. In: *SIAM J. Matrix Anal. Appl.* 29 (2007), pp. 434–454.

[53] Othmar Koch and Christian Lubich. 'Dynamical Tensor Approximation'. In: *SIAM J. Matrix Analysis Applications* 31 (Jan. 2010), pp. 2360–2375. DOI: 10.1137/09076578X.

[54] Tamara G. Kolda and Brett W. Bader. 'Tensor Decompositions and Applications'. In: *SIAM Review* 51.3 (2009), pp. 455–500. DOI: 10.1137/07070111X. eprint: https://doi.org/10.1137/07070111X. URL: https://doi.org/10.1137/07070111X.

[55] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar and Maja Pantic. 'TensorLy: Tensor Learning in Python'. In: *Journal of Machine Learning Research (JMLR)* 20.26 (2019).

[56] Alex Krizhevsky. 'Learning Multiple Layers of Features from Tiny Images'. In: *University of Toronto* (May 2012).

[57] Siddharth Krishna Kumar. *A general metric for identifying adversarial images*. 2018. DOI: 10.48550/ARXIV.1807.10335. URL: https://arxiv.org/abs/1807.10335.

[58] Alexey Kurakin, Ian Goodfellow and Samy Bengio. *Adversarial examples in the physical world*. 2016. DOI: 10.48550/ARXIV.1607.02533. URL: https://arxiv.org/abs/1607.02533.

[59] Yann LeCun, Y. Bengio and Geoffrey Hinton. 'Deep Learning'. In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[60] J. Lee and J.M. Lee. *Manifolds and Differential Geometry*. Graduate studies in mathematics. American Mathematical Society, 2009. ISBN: 9780821848159. URL: https://books.google.no/books?id=QqHdHy9WsEoC.

[61] Mario Lezcano-Casado and David Martínez-Rubio. *Cheap Orthogonal Constraints in Neural Networks: A Simple Parametrization of the Orthogonal and Unitary Group*. 2019. DOI: 10.48550/ARXIV.1901.08428. URL: https://arxiv.org/abs/1901.08428.

[62] Qianxiao Li, Long Chen, Cheng Tai and Weinan E. *Maximum Principle Based Algorithms for Deep Learning*. 2018. arXiv: 1710.09513 [cs.LG].

[63] Haw-minn Lu, Yeshaiahu Fainman and Robert Hecht-Nielsen. 'Image manifolds'. In: *Electronic Imaging*. 1998.

[64] Christian Lubich, Thorsten Rohwedder, Reinhold Schneider and Bart Vandereycken. 'Dynamical Approximation by Hierarchical Tucker and Tensor-Train Tensors'. In: *SIAM Journal on Matrix Analysis and Applications* 34.2 (2013), pp. 470–494. DOI: 10.1137/120885723. eprint: https://doi.org/10.1137/120885723. URL: https://doi.org/10.1137/120885723.

[65] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu and Andrew Ng. 'Reading Digits in Natural Images with Unsupervised Feature Learning'. In: *NIPS* (Jan. 2011).

[66] E. Oja. 'Data Compression, Feature Extraction, and Autoassociation in Feedforward Neural Networks'. In: *Artificial Neural Networks*. Ed. by T. Kohonen, K. Mäkisara, O. Simula and J. Kangas. Vol. 1. Elsevier Science Publishers B.V., North-Holland, 1991, pp. 737–745.

[67] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703.

[68] D. Pedoe. *Geometry: A Comprehensive Course*. Dover books on advanced mathematics. Dover Publications, 1988. ISBN: 9780486658124. URL: https://books.google.no/books?id=-U5TyIw15rUC.

[69] Robert Pless and Richard Souvenir. 'Manifold learning for natural image sets'. In: 2006.

[70] Hang Shao, Abhishek Kumar and P. Thomas Fletcher. *The Riemannian Geometry of Deep Generative Models*. 2017. DOI: 10.48550/ARXIV.1711.08014. URL: https://arxiv.org/abs/1711.08014.

[71] Eduardo Sontag. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems*. Jan. 1998. DOI: 10.1007/978-1-4612-0577-7.

[72] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow and Rob Fergus. *Intriguing properties of neural networks*. 2013. DOI: 10.48550/ARXIV.1312.6199. URL: https://arxiv.org/abs/1312.6199.

[73] Matus Telgarsky. *Benefits of depth in neural networks*. 2016. arXiv: 1602.04485 [cs.LG].

[74] Joshua B. Tenenbaum, Vin de Silva and John C. Langford. 'A Global Geometric Framework for Nonlinear Dimensionality Reduction'. In: *Science* 290.5500 (2000), pp. 2319–2323. DOI: 10.1126/science.290.5500.2319. eprint: https://www.science.org/doi/pdf/10.1126/science.290.5500.2319. URL: https://www.science.org/doi/abs/10.1126/science.290.5500.2319.

[75] Matthew Thorpe and Yves van Gennip. *Deep Limits of Residual Neural Networks*. 2020. arXiv: 1810.11741 [math.CA].

[76] Robert Tibshirani. 'Regression Shrinkage and Selection via the Lasso'. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 00359246. URL: http://www.jstor.org/stable/2346178.

[77] Yusuke Tsuzuku, Issei Sato and Masashi Sugiyama. *Lipschitz-Margin Training: Scalable Certification of Perturbation Invariance for Deep Neural Networks*. 2018. DOI: 10.48550/ARXIV.1802.04034. URL: https://arxiv.org/abs/1802.04034.

[78] Ledyard R. Tucker. 'Some mathematical notes on three-mode factor analysis'. In: *Psychometrika* 31.3 (1966), pp. 279–311. DOI: 10.1007/BF02289464. URL: https://doi.org/10.1007/BF02289464.

[79] Han Xiao, Kashif Rasul and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. DOI: 10.48550/ARXIV.1708.07747. URL: https://arxiv.org/abs/1708.07747.

[80] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Bhat and Xiaolin Li. 'Adversarial Examples: Attacks and Defenses for Deep Learning'. In: (Dec. 2017).

[81] Zhenyue Zhang and Hongyuan Zha. *Principal Manifolds and Nonlinear Dimension Reduction via Local Tangent Space Alignment*. 2002. DOI: 10.48550/ARXIV.CS/0212008. URL: https://arxiv.org/abs/cs/0212008.

[82] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman and Alexei A. Efros. *Generative Visual Manipulation on the Natural Image Manifold*. 2016. DOI: 10.48550/ARXIV.1609.03552. URL: https://arxiv.org/abs/1609.03552.

Appendix A

Appendices

A.1 The Cayley map

For the efficient implementation of the network, we need to address the computation of analytic functions of matrices $B \in \mathbb{R}^{m\times m}$ given in the factorised form

$$B = CD^T, \quad C \in \mathbb{R}^{m\times p}, \quad D \in \mathbb{R}^{m\times p}.$$

We are particularly interested in

$$\mathrm{cay}(B) = \left(I - \frac{1}{2}B\right)^{-1}\left(I + \frac{1}{2}B\right).$$

We want to exploit the fact that $B = CD^T$, is factorised and obtain efficient implementations of cay, especially under the assumption that p is small. Notice that in our numerical methods we compute the Cayley transformation of $B = CD^T$

$$C = [F_U,\, -U], \quad D = [U,\, F_U],$$

with $p = 2k$, and with $U^TU = I$, $F_U^TU = 0$.

Notice that for analytic functions $\phi(B) = \sum_{i=0}^{\infty}\alpha_i B^i$ we have

$$\phi(B) = \alpha_0 I + C\sum_{i=1}^{\infty}\alpha_i (D^TC)^{i-1}\,D^T = \alpha_0 I + C\,\left.\frac{\phi(z)-1}{z}\right|_{z=D^TC} D^T,$$

and in our case

$$D^TC = [U,\, F_U]^T[F_U,\, -U] = \begin{bmatrix} O & -I \\ F_U^TF_U & O \end{bmatrix}.$$

When $\phi(z) = \text{cay}(z) = \frac{1+\frac{z}{2}}{1-\frac{z}{2}}$, $\frac{\phi(z)-1}{z} = (1-\frac{1}{2}z)^{-1}$ so

$$\text{cay}(CD^T) = I + C(I - \tfrac{1}{2}D^TC)^{-1}D^T.$$

For an alternative method consider the QR-factorisation of D,

$$D = [U,\, F_U] = [U,\, U^\perp]\begin{bmatrix} I & O \\ O & R_{2,2} \end{bmatrix},$$

where $[U,\, U^\perp]$ has $2k$ orthonormal columns, and $F_U = U^\perp R_{2,2}$ is the QR factorisation of F_U. From this factorisation we can construct a useful factorisation of C:

$$C = [F_U,\, -U] = [U,\, F_U]\begin{bmatrix} O & -I \\ I & O \end{bmatrix} = [U,\, U^\perp]\begin{bmatrix} O & -I \\ R_{2,2} & O \end{bmatrix}$$

By multiplying together the two decomposed factors we obtain

$$CD^T = [U,\, U^\perp]\begin{bmatrix} O & -R_{2,2}^T \\ R_{2,2} & O \end{bmatrix}[U,\, U^\perp]^T,$$

and from this it is possible to show that

$$\text{cay}(CD^T) = I + [U,\, U^\perp]G[U,\, U^\perp]^T.$$

$$G = \begin{bmatrix} O & -R_{2,2}^T \\ R_{2,2} & O \end{bmatrix}\left(I - \frac{1}{2}\left(\begin{bmatrix} O & -R_{2,2}^T \\ R_{2,2} & O \end{bmatrix}\right)\right)^{-1}$$

A.2 The MNIST data set

Top row of Figure A.1 shows samples from the original MNIST data set. The Second row shows truncated SVD performed with $k = 2$. The third and bottom row shows truncated SVD with $k = 3$. There is a slight difference between the compressed images. The number 7 has lost much shape when $k = 2$. It could be, perhaps in other similar cases, be difficult to distingush the number 7 from the number 0. Therefore, as the compression is still quite significant if choosing $k = 3$, , and as we want to keep the most important singular values 4.2, this is our chosen truncation.

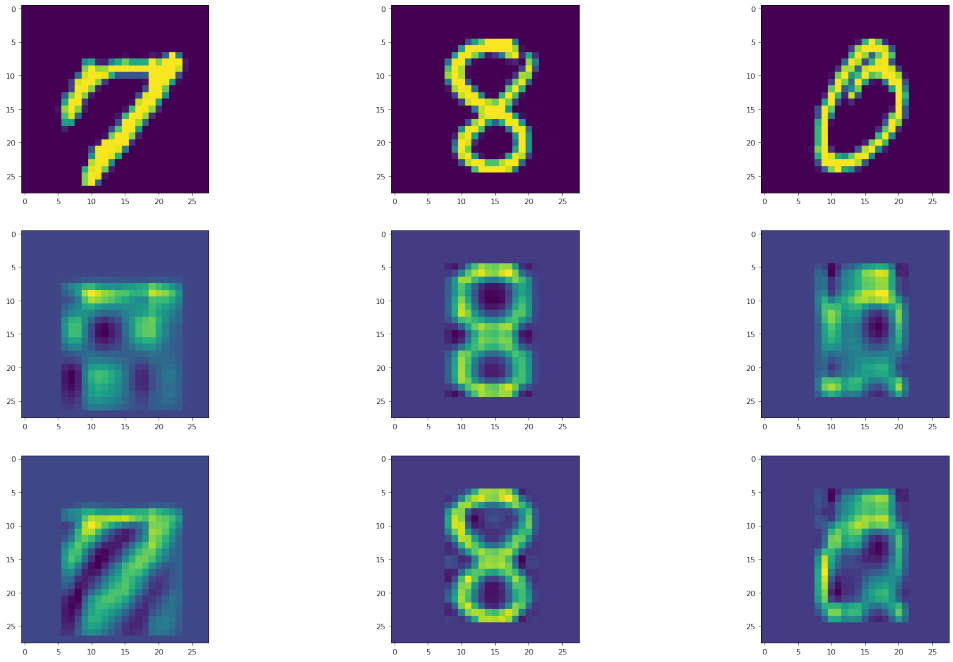


Figure A.1: Top row: Original samples of MNIST data set
Middle row: Compressed images using truncated SVD of order $k = 2$.
Bottom row: Compressed images using truncated SVD of order $k = 3$.

A.3 The FashionMNIST data set

Top row of Figure A.1 shows samples from the original FashionMNIST data set. The Second row shows truncated SVD performed with $k = 2$. The third and bottom row shows truncated SVD with $k = 3$. The difference between the compressed images is almost not visible. The main difference is the dress the the left, one can see the long sleeves more pronounced, which makes the difference between the dress and the t-shirt to the right the most visible. Therefore, visually it makes more sense to choose truncation $k = 3$, and as we want to keep the most important singular values 4.2, this is our chosen truncation.

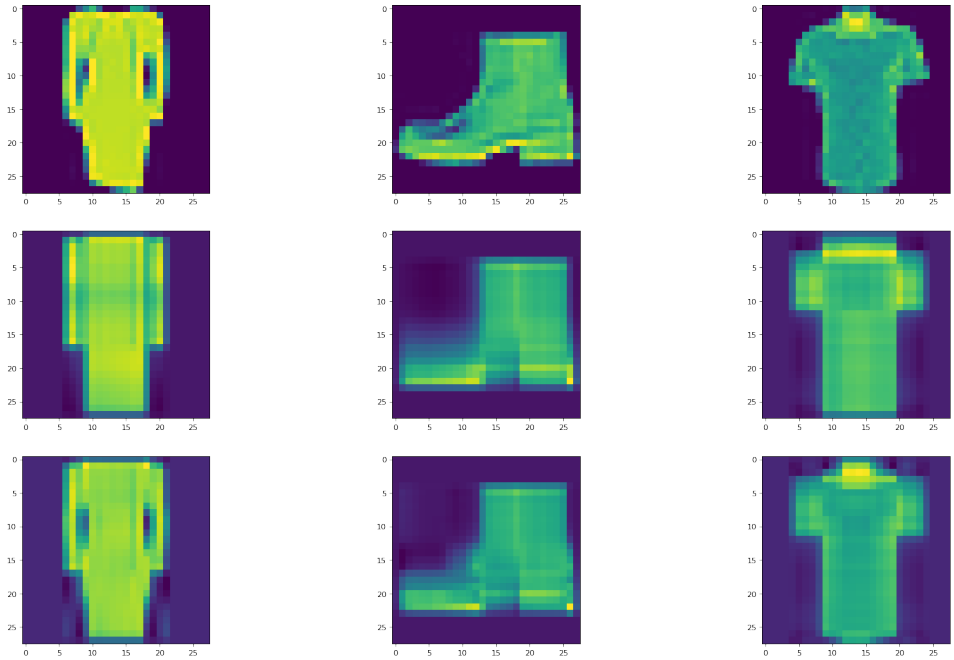


Figure A.2: Top row: Original samples of FashionMNIST data set
Middle row: Compressed images using truncated SVD of order $k = 2$.
Bottom row: Compressed images using truncated SVD of order $k = 3$.

A.4 The CIFAR10 data set

Top row of Figure A.3 shows samples from the original CIFAR10 data set. The Second row shows compressed images where Tucker Decomposition has been performed with Tucker rank $r = [3, 3, 3]$. The third and bottom row shows compressed images using Tucker Decomposition of Tucker rank $r = [3, 9, 9]$. The difference between the compressed images is almost not visible. Only slight difference in pigmentation in certain cells. Therefore, visually we cannot determine from these sample images which rank we should choose. We will choose a truncation based upon the singular values, see Figure 4.2, and the error in the Tucker Decomposition , see Figure 4.3(a).

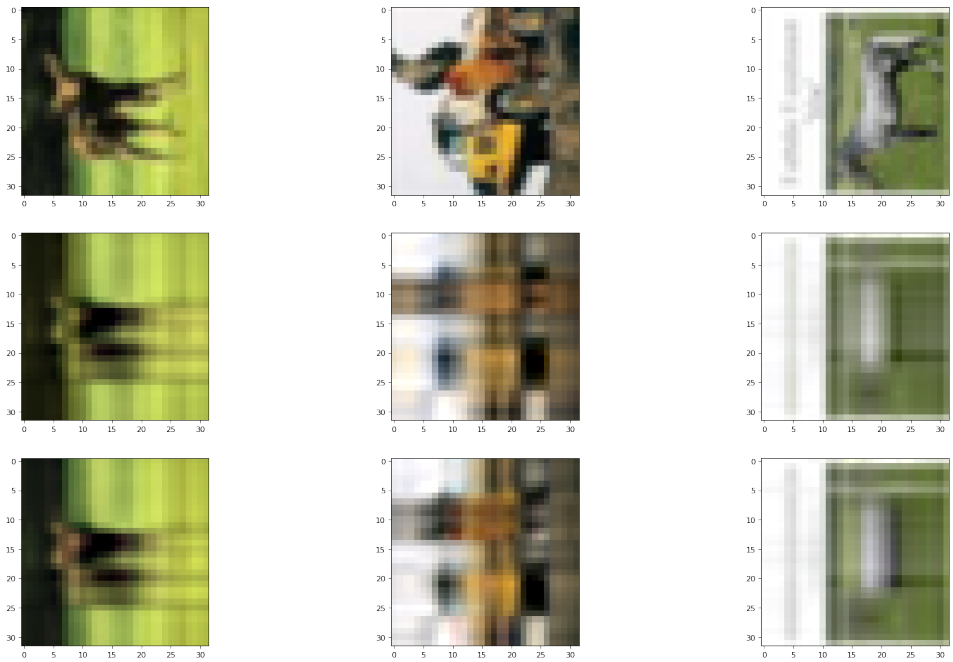


Figure A.3: Top row: Original samples of the CIFAR10 data set
Middle row: Compressed images using Tucker Decomposition of rank $r = [3, 3, 3]$.
Bottom row: Compressed images using Tucker Decomposition of rank $r = [3, 9, 9]$.

A.5 The SVHN data set

Top row of Figure A.4 shows samples from the original SVHN data set. The Second row shows compressed images where Tucker Decomposition has been performed with Tucker rank $r = [3, 3, 3]$. The third and bottom row shows compressed images using Tucker Decomposition of Tucker rank $r = [3, 9, 9]$. The difference between the compressed images is almost not visible. The main difference is the dress the the left, one can see the long sleeves more prononced, which makes the difference between the dress and the tshirt to the right the most visible. We will choose a truncation based upon the singular values, see Figure 4.2, and the error in the Tucker Decomposition , see Figure 4.3(b).

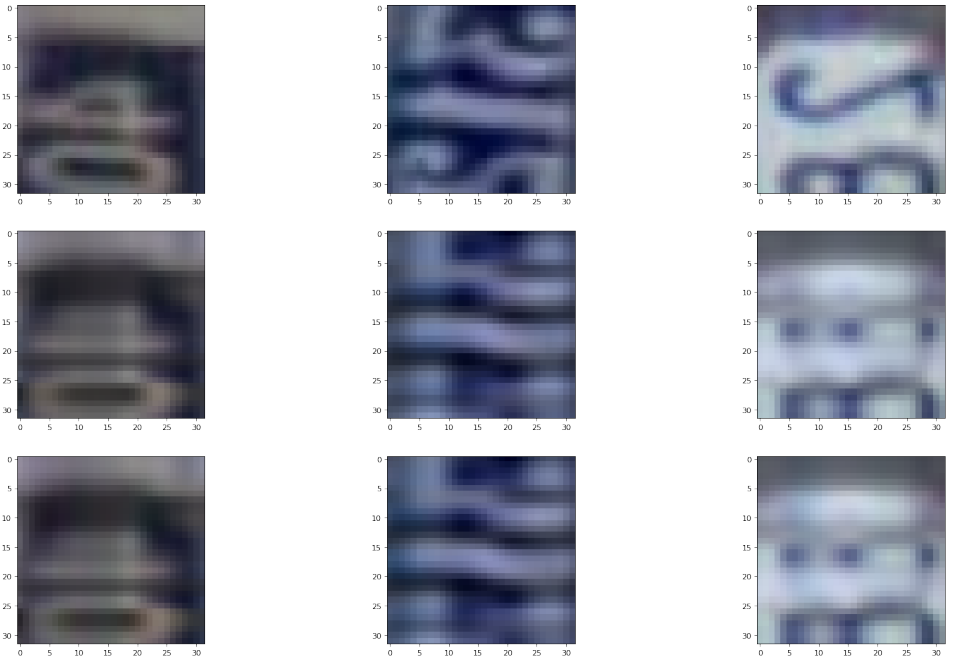
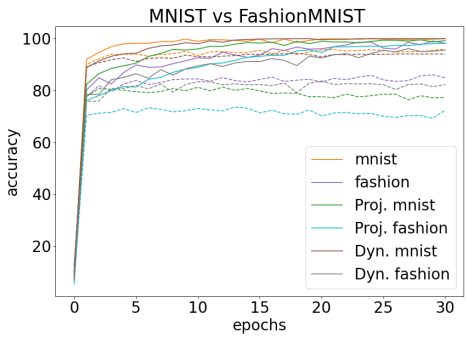
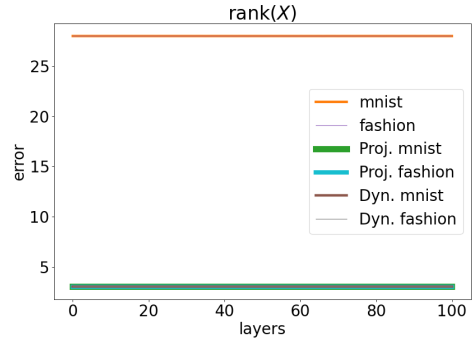


Figure A.4: Top row: Original samples of the SVHN data set
Middle row: Compressed images using Tucker Decomposition of rank $r = [3, 3, 3]$.
Bottom row: Compressed images using Tucker Decomposition of rank $r = [3, 9, 9]$.

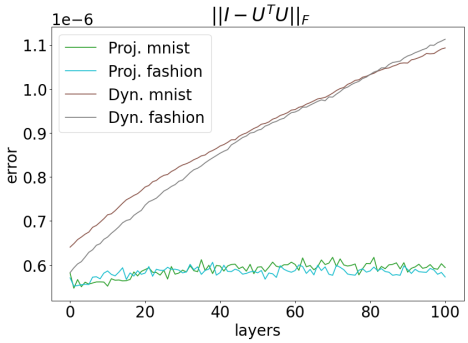
A.6 Summary of results of MNISTvsFashionMNIST on deep networks.



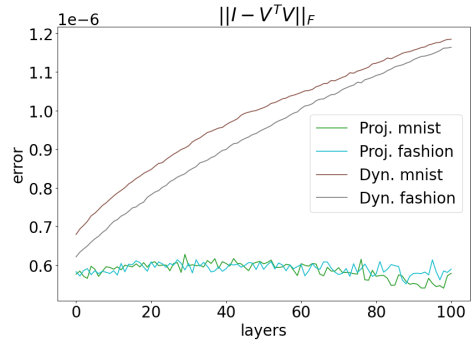
(a) Convergence of the deep $L = 100$ networks on different data sets.



(b) Average rank of the output for each layer in the trained deep networks.



(c) Average error in orthogonality in U at every layer of the relevant trained networks



(d) Average error in orthogonality in V at every layer of the relevant trained networks

Figure A.5: Summary of results of training deep networks $L = 100$ on MNIST and FashionMNIST data sets.

Camilla Balestrand Klemetsen

Neural Networks on Low-Rank and Stiefel Manifolds

NTNU
Norwegian University of
Science and Technology