Skolkovo Institute of Science and Technology

APPLICATIONS OF DIFFERENTIAL EQUATIONS AND REDUCED-ORDER MODELING FOR DEEP LEARNING

Doctoral Thesis

by

TALGAT DAULBAEV

DOCTORAL PROGRAM IN COMPUTATIONAL AND DATA SCIENCE AND ENGINEERING

Supervisor:
Ivan Oseledets, Full Professor

Co-supervisor:
Andrzej Cichocki, Full Professor

Moscow — 2023

I hereby declare that the work presented in this thesis was carried out by myself at Skolkovo Institute of Science and Technology, Moscow, except where due acknowledgement is made, and has not been submitted for any other degree.

Talgat Daulbaev
Professor Ivan Oseledets
Professor Andrzej Cichocki

Abstract

In this thesis, we explore the various ways in which differential equations and reduced-order modeling can be applied in the context of deep learning. The work is divided into two main parts.

The first part is about neural ordinary differential equations, a type of neural network architecture that uses systems of ordinary differential equations as a fundamental building block. We propose a new interpolation-based algorithm for efficient and memory-effective training of these networks, examine the influence of different normalization layers on the performance of neural ODEs, and demonstrate how the parametrization of the training solver can affect the robustness of these networks.

The final two chapters investigate the application of classic techniques from differential equations to standard neural network architectures. Specifically, we explore the use of reduced-order modeling techniques to accelerate artificial neural networks, and the application of the active subspace method for compression of these networks and the creation of universal adversarial attack vectors.

Publications

The body of this thesis is formed by four published papers and a single preprint. A sign ⋆ denotes equal contribution. All papers and major contributions of the author are as follows.

Chapter 2 expands on:

• Talgat Daulbaev, Alexander Katrutsa, Larisa Markeeva, Julia Gusak, Andrzej Cichocki, and Ivan Oseledets
"Interpolation technique to speed up gradients propagation in neural ODEs"
Advances in Neural Information Processing Systems, 33, pages 16689-16700, 2020

Author's contributions in the paper include: design, implementation, theoretical analysis, experiments, draft, and revisions.

Chapter 5 expands on:

• Julia Gusak⋆, Talgat Daulbaev⋆, Evgeny Ponomarev, Andrzej Cichocki, and Ivan Oseledets "Reduced-order Modeling of Deep Neural Networks"
Computational Mathematics and Mathematical Physics, 61(5), pages 774-785, 2021

Author's contributions in the paper include: design, implementation, theoretical analysis, experiments, draft, and revisions.

Chapter 3 expands on:

• Julia Gusak, Larisa Markeeva, Talgat Daulbaev, Alexander Katrutsa, Andrzej Cichocki, and Ivan Oseledets "Towards Understanding Normalization in Neural ODEs"
ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations

Author's contributions in the paper include: theoretical analysis, draft, and revisions.

Chapter 6 expands on:

• Chunfeng Cui, Kaiqi Zhang, Talgat Daulbaev, Julia Gusak, Ivan Oseledets, and Zheng Zhang "Active Subspace of Neural Networks: Structural Analysis and Universal Attacks"
SIAM Journal on Mathematics of Data Science, 2(4), pages 1096-1122.

Author's contributions in the paper include: implementation and experiments.

Chapter 4 expands a pre-print:

- Julia Gusak, Talgat Daulbaev, Alexander Katrutsa, Andrzej Cichocki, and Ivan Oseledets “Meta-solver for neural ordinary differential equations” arXiv preprint arXiv:2103.08561.

Author’s contributions in the paper include: experiments and ideas.

Acknowledgements

I want to thank everyone who has helped me throughout my journey to graduate school and the creation of this thesis. The list of these people is so huge that it would hardly fit on a page.

I am incredibly grateful to my supervisors, Prof. Ivan Oseledets and Prof. Andrzej Cichocki. They are always supportive and kind, and my gratitude to them can not be overestimated. In particular, Prof. Ivan Oseledets, who was also a supervisor for my master's thesis, taught me a tremendous amount of things and showed me how research should be done.

I would like to thank my co-authors (in alphabetical order) Alexander Katrutsa, Chunfeng Cui, Evgeny Ponomarev, Julia Gusak, and Larisa Markeeva.

Also, I want to say "thank you" to all my teachers from my school, Moscow State University, and Skoltech. Especially, I am grateful to my first math teachers Boris V. Antonov and Oleg I. Yuzhakov, and to my first supervisor, the late Konstantin V. Rudakov.

Finally, I thank my family (especially my mum for her constant support) and friends. Without them, this thesis would not exist.

Contents

Abstract ii

Publications iii

Acknowledgements v

List of Figures ix

List of Tables xii

Notation xiv

Introduction 1

1 Preliminaries 3
- 1.1 Considered Machine Learning Problems . . . 3
- 1.2 Neural Ordinary Differential Equations . . . 5
  - 1.2.1 Adaptive Runge–Kutta Solvers . . . 6
- 1.3 Discrete Empirical Interpolation Method . . . 10
- 1.4 Active Subspace Method . . . 11

2 Acceleration of Gradients Propagation in Neural ODEs 12
- 2.1 Introduction . . . 12
- 2.2 Related Work . . . 14
- 2.3 Interpolated Reverse Dynamic Method . . . 14
- 2.4 Upper Bound on the Gradient Error Induced by Interpolated Activations . . . 17
- 2.5 Numerical Experiments . . . 20
  - 2.5.1 Experimental settings . . . 21
    - 2.5.1.1 Classification . . . 21
    - 2.5.1.2 Density estimation . . . 21
    - 2.5.1.3 VAE . . . 22
  - 2.5.2 Density Estimation . . . 23
  - 2.5.3 Variational Autoencoder . . . 24
  - 2.5.4 Classification . . . 25
  - 2.5.5 Number of Chebyshev Grid Points . . . 25
- 2.6 Conclusion . . . 27

3 Towards Understanding Normalization in Neural ODEs **28**
3.1 Introduction . . . 28
3.2 Background . . . 28
3.3 Numerical Experiments . . . 29
3.3.1 Accuracy . . . 30
3.3.2 $(\mathcal{S}, n)$-criterion of dynamics smoothness in the trained model . . . 30
3.4 Discussion and Further research . . . 33

4 Exploring Robustness of Different Solvers for Neural ODEs **34**
4.1 Introduction . . . 34
4.1.1 Related works . . . 35
4.2 Meta Neural ODE . . . 37
4.3 Experiments . . . 38
4.3.1 Motivation to explore solver parameterizations . . . 38
4.3.2 Adversarial training on CIFAR-10 . . . 39
4.3.3 Neural Networks attack Neural ODEs . . . 40
4.3.4 Neural ODEs attack Neural ODEs . . . 40
4.4 Conclusion . . . 40

5 Reduced-Order Modeling of Deep Neural Networks **44**
5.1 Introduction . . . 44
5.2 Background . . . 45
5.2.1 Maximum Volume Algorithm and Sketching . . . 45
5.2.2 Computation of Low-Dimensional Embeddings . . . 46
5.3 Method . . . 46
5.3.1 A Toy Example: MLP . . . 47
5.3.2 Convolutional Neural Networks . . . 48
5.3.3 Residual Networks . . . 49
5.3.4 Approximation error . . . 49
5.4 Experiments . . . 50
5.4.1 Singular values . . . 50
5.4.2 Fully-connected networks . . . 50
5.4.3 Convolutional networks . . . 52
5.4.4 Comparisons with other approaches . . . 53
5.5 Discussion . . . 54
5.6 Related work . . . 56
5.7 Conclusion . . . 57

6 Active Subspaces for Neural Networks **58**
6.1 Introduction . . . 58
6.1.1 Contributions . . . 59
6.2 Active Subspace . . . 60
6.2.1 Response Surface . . . 61
6.3 Active Subspace for Structural Analysis and Compression of Deep Neural Networks . . . 62
6.3.1 Deep Neural Networks . . . 62
6.3.2 The Number of Active Neurons . . . 63

6.3.3 Active Subspace Network (ASNet) 65
6.3.4 The Active Subspace Layer 66
6.3.5 Polynomial Chaos Expansion Layer 67
6.3.6 Structured Re-training of ASNet 70
6.4 Active-Subspace for Universal Adversarial Attacks 71
6.4.1 Universal Perturbation of Deep Neural Networks . . . 72
6.4.2 Recursive Projection Method 72
6.5 Numerical Experiments 73
6.5.1 Structural Analysis and Compression 73
6.5.1.1 Choices of Parameters 75
6.5.1.2 Efficiency of the ASNet 75
6.5.1.3 CIFAR-10 76
6.5.1.4 CIFAR-100 77
6.5.2 Universal Adversarial Attacks 77
6.5.2.1 Fashion-MNIST 78
6.5.2.2 CIFAR-10 79
6.5.2.3 CIFAR-100 80
6.6 Conclusions and Discussions 80

Conclusions **85**

Bibliography **86**

List of Figures

2.1 Example of potential instability of RDM as described in [55]. We took a single ResNet-block with random weights as a right-hand side and solved the initial value problem 2.1 with an initial condition z_0 (left) and obtained the image z_1 (middle). After that, we integrated the same initial value problem backward-in-time and obtained $\widehat{z}_0$ (right). Images on the left and on the right should coincide in order to perform an accurate backward pass, but they obviously do not. 13

2.2 Comparison of different schemes to make forward and backward passes through the ODE block. Red circles indicate that the activations are stored at these time points. Red arrows indicate that during ODE steps, the outputs of the intermediate layer are stored to propagate gradients. Green arrows correspond to the steps with ODE solvers. Blue arrows correspond to the steps with automatic differentiation through the stored computational graph. Activations in Chebyshev grid points (t_0, τ_1, τ_2 and t_1 in Figure 2.2c) are stored in the interpolation approach during the forward pass. Chebyshev grid points do not necessarily coincide with time steps of ODE solver, but activations in these points can be recovered from the computed activations with ODE solver. The stored activations are used to approximate activations in the backward pass. The dotted arrows in Figure 2.2c shows that activations in t_0, τ_1, τ_2 and t_1 are used to interpolate activations in the backward pass. . . . 18

2.3 The dependence of the IRDM gradients error in ℓ_1-norm with respect to the number of nodes in the Chebyshev grid and the tolerance of the DOPRI5 method. The output of the standard backpropagation performed for the DOPRI5 with `1e-7` tolerance was used as a ground truth. 23

2.4 Comparison of the IRDM with the RDM (baseline from FFJORD) on density estimation problem for tabular dataset `miniboone`. The number of points in the Chebyshev grid N used in the IRDM is given in the legend. 24

2.5 Total number of $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ evaluations for density estimation datasets. 24

2.6 Comparison of the number of right-hand side evaluations for the IRDM and the RDM in training variational autoencoder. . 24

2.7 Experiments results in the image classification task. The reported values are averaged over three trained models corresponding to the considered tasks. 25

2.8 Comparison of IRDM (our method) and RDM on density estimation problem for toy datasets `2spirals`, `pinwheel`, `moons`, and `circles` in terms of test loss versus wall-clock training time. Comparison results for every dataset are presented in the corresponding subplot. The number of points in Chebyshev grid N used in the IRDM is given in legend. 26
2.9 Comparison of IRDM (our method) and RDM on density estimation problem for toy datasets `2spirals`, `pinwheel`, `moons`, and `circles` in terms of total number of the right-hand side evaluations versus number of iterations. Comparison results for every dataset are presented in the corresponding subplot. The number of points in Chebyshev grid N used in the IRDM is given in legend. 27

3.1 Illustration of how the choice of ODE solver and normalizations during training implicitly affects the smoothness of learned dynamics. Each subplot corresponds to the model trained with a fixed ODE solver and normalization scheme. Models within one row have the same type of training solver ((Euler, n), $n = 2, 16, 32$ from top to bottom). Models within one column have the same normalization technique. For example, subplot in the third row and the second column corresponds to the ODENet4 model trained with (Euler, 32) solver with BN after the first convolutional layer and LN after convolutional layers inside ODE block. Lines of different style corresponds to different types of test solvers. If model accuracy does not drop when the more powerful ODE solver is used, we conclude that, according to $(\mathcal{S}, n)$-criterion, the model provides a smooth dynamics. For example, the model (Euler, 32) BN-LN trains a smooth dynamics, while (Euler, 2) BN-LN fails to do that. Also, we can observe that to learn the smooth dynamics during training, for some normalization schemes less powerful solvers are required. If we compare BN-LN and BN-WN models, we can see that the first one learns smooth dynamics when Euler with $n = 16$ is used, but the latter one does that only for $n = 32$. 32

4.1 Robust accuracy of the model on MNIST dataset vs. different values of parameter u in the 2-nd order Runge-Kutta solver (see Table 4.1). 39

5.1 We plot singular values of all layers for CIFAR-10 for VGG-19 (left) and ResNet-56 (right). Each singular value is divided by the largest one for this layer. One can see that most singular values are relatively small. 52
5.2 RON for different LeNet models. 52
5.3 Accuracy and FLOP reduction for RON accelerated models on CIFAR-10. 54

6.1 Structural analysis of deep neural networks by the active subspace (AS). All experiments are conducted on CIFAR-10 by VGG-19. (a) The number of neurons can be significantly reduced by the active subspace. Here, the number of active neurons is defined by Definition 6.3.1 with a threshold $\epsilon = 0.05$; (b) Most of the parameters are distributed in the last few layers; (c) The active subspace direction can perturb the network significantly. 60
6.2 (a) The original deep neural network; (b) The proposed ASNet with three parts: a pre-model, an active subspace (AS) layer, and a polynomial chaos expansion (PCE) layer. 66
6.3 Distribution of the first two active subspace variables at the 6-th layer of VGG-19 for CIFAR-10. 68
6.4 Perturbations along the directions of an active-subspace direction and of principal component, respectively. (a) The function $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} - b$. (b) The perturbed function along the active-subspace direction. (c) The perturbed function along the principal component analysis direction. 71
6.5 Structural analysis of VGG-19 on the CIFAR-10 dataset. (a) The first 200 singular values for layers $4 \leq l \leq 7$; (b) The accuracy (without any fine-tuning) obtained by active-subspace (AS) and polynomial chaos expansions (PCE) compared with principal component analysis (PCA) and logistic regression (LR). 76
6.6 Universal adversarial attacks for the Fashion-MINST with respect to different ℓ_2-norms. (a)-(c): the results for attacking one class dataset. (d)-(f): the results for attacking the whole dataset. 81
6.7 The effect of our attack method on one data sample in the Fashion-MNIST dataset. (a) A trouser from the original dataset. (b) An active-subspace perturbation vector with the ℓ_2 norm equals 5. (c) The perturbed sample is misclassified as a t-shirt/top by the deep neural network. 82
6.8 Universal adversarial attacks of VGG-19 on CIFAR-10 with respect to different ℓ_2-norm perturbations. (a)-(c): The training attack ratio, the testing attack ratio, and the CPU time in seconds for attacking one class dataset. (d)-(f): The results for attacking ten classes dataset together. 83
6.9 Adversarial attack of VGG-19 on CIFAR-10 with different number of training samples. The ℓ_2-norm perturbation is fixed as 10. (a) The results of attacking the dataset from the first class; (b) The results of attacking the whole dataset with 10 classes. 83
6.10 Results for universal adversarial attack for CIFAR-100 with respect to different ℓ_2-norm perturbations. (a)-(c): The results for attacking the dataset from the first class. (d)-(f): The results for attacking ten classes dataset together. 84

List of Tables

2.1 Time (in seconds) to perform 10000 training iterations for toy datasets. 26

3.1 Comparison of normalization techniques for ODENet10 architecture on CIFAR-10. BN – batch normalization, LN – layer normalization, WN – weight normalization, SN – spectral normalization, NF – the absence of any normalization. To perform back-propagation, we exploit ANODE with a non-adaptive ODE solver. Namely, we use Euler scheme with $N_t = 8$, where N_t is a number of time steps used to solve IVP (3.1). The first row corresponds to the normalization in the ODE blocks. We use BN after the first convolutional layer and inside ResNet blocks, respectively. Standard ResNet10 architecture (only ResNet blocks are used) gives **0.931** test accuracy. 31

4.1 2-stage RK method of the 2-nd order 37
4.2 Blackbox attacks on CIFAR10 Neural ODE models. Source models are from RobustBench [32] and papers by Carmon et al. [18], Sehwag et al. [152], and Wong, Rice, and Kolter [171]. PGD attack is performed with 20 iterations and DeepFool attack is performed with 50 iterations. 42
4.3 Greybox attacks for $\varepsilon = 8/255$. Mean robust accuracy and standard errors averaged over three runs are reported below. Parameters for PGD attack are the following: number of steps is 7, step size is $2/255$. Maximum number of steps in DeepFool attack is 50. 43
4.4 Comparison of adversarial robustness of the RK2 and Euler solvers. 43

5.1 Accuracy and FLOP trade-off for the models accelerated with RON on CIFAR-10 dataset. DCP is a channel pruning method from [186]. 54
5.2 VGG on CIFAR-100. RON $N\times$ stands for the accelerated model, where feature dimentionality of last layers is reduced by $N\times$ times comparing to the teacher. 55
5.3 VGG on SVHN. RON $N\times$ stands for the accelerated model, where feature dimentionality of last layers is reduced by $N\times$ times comparing to the teacher. 55

5.4 Comparison of acceleration methods for VGG-19 on CIFAR-10. Pre-trained baseline has 93.7% accuracy. The higher FLOP reduction the better. The smaller accuracy drop the better. . . 55

6.1 Comparison of number of neurons r of VGG-19 on CIFAR-10. For the storage speedup, the higher is better. For the accuracy reduction before or after fine-tuning, the lower is better. 75

6.2 Accuracy and storage on VGG-19 for CIFAR-10. Here, “Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, “AS" and “PCE" denote the active subspace and polynomial chaos expansion layer, respectively. 77

6.3 Accuracy and storage on ResNet-110 for CIFAR-10. Here, “Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, “AS" and “PCE" denote the active subspace and polynomial chaos expansion layer, respectively. 78

6.4 Accuracy and storage on VGG-19 for CIFAR-100. Here, “Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, “AS" and “PCE" denote the active subspace and polynomial chaos expansion layer, respectively. 79

6.5 Accuracy and storage on ResNet-110 for CIFAR-100. Here, “Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, “AS" and “PCE" denote the active subspace and polynomial chaos expansion layer, respectively. 80

6.6 Cross-model performance for CIFAR-10 80

6.7 Summary of the universal attack for different datasets by the active-subspace compared with UAP and the random vector. The norm of perturbation is equal to 10. 81

Notation

AS . . . . . .Active Subspaces

BLI . . . . . .Barycentric Lagrange Interpolation

BN . . . . . .Batch Normalization

CNN . . . . . .Convolutional Neural Networks

CP . . . . . .Channel Pruning

CPU . . . . . .Central Processing Unit

DCP . . . . . .Discrimination-aware Channel Pruning

DEIM . . . . . .Discrete Empirical Interpolation Method

DOPRI5 . . . . . .Dormand-Prince of Order (4)5

ELBO . . . . . .Evidence Lower Bound

FFJORD . . . . . .Free-form Jacobian of Reversible Dynamics

FGSM . . . . . .Fast Gradient Sign Method

FLOP . . . . . .Floating Point Operation

FOOC . . . . . .First-Order Optimality Conditions

IRDM . . . . . .Interpolated Reverse Dynamic Method

IVP . . . . . .Initial Value Problem

LN . . . . . .Layer Normalization

MaxVol . . . . . .Maximum Volume Algorith

MLP . . . . . .Multilayer Perceptron

ODE . . . . . .Ordinary Differential Equations

PCE . . . . . .Polynomial Chaos Expansion

PDE . . . . . .Partial Differential Equations

PGD . . . . . .Projected Gradient Descent Attack

POD . . . . . .Proper Orthogonal Decomposition

RDM . . . . . .Reverse Dynamic Method

ReLU . . . . . .Rectified Linear Unit

ResNet . . . . . .Residual Networks

RK . . . . . .Runge–Kutta methods

RON . . . . . .Reduced Order Network

SGD . . . . . .Stochastic Gradient Descent

SN . . . . . .Spectral Normalization

SVD . . . . . .Singular Value Decomposition

UAP . . . . . .Universal Adversarial Perturbations

VAE . . . . . .Variational Autoencoders

VGG . . . . . .Very Deep Convolutional Networks

Introduction

Motivation

The study of machine learning and its connections to various areas of mathematics and physics has led to a number of significant findings, particularly in the realm of differential equations. These equations, which can be ordinary, partial, or stochastic in nature, play a critical role in machine learning tasks. For example, gradient descent algorithms can be viewed as numerical methods for solving differential equations [4, 115].

differential equations. These connections can be found in both classic and more recent studies of the field. For instance, residual networks [74] have incorporated elements of differential equations in their design, such as the use of Euler's method to solve special ordinary differential equations and the incorporation of the Runge-Kutta method. Additionally, the hierarchical structure of feature maps in residual networks has been inspired by the multigrid method for solving partial differential equations [43, 168].

However, the influence of differential equations on deep learning goes beyond simple analogies and discretizations. One example of this is the class of neural networks known as neural ordinary differential equations (ODEs)[22]. These networks consist of ODE blocks that output solutions to differential equations at given times, with the parameters of the ODE blocks being encapsulated in their right-hand sides. Neural ODEs have been applied to a variety of problems, including classification[22], time series prediction [95], video generation [130], and generative modeling [61].

Another promising application of differential equations in deep learning is the use of diffusion models [163, 94]. These models are a type of ordinary and stochastic differential equation. Model reduction techniques, which are often used to reduce the computational cost and memory requirements of large systems of ODEs, are also closely related to differential equations.

In this thesis, we continue to explore the ways in which differential equations can be applied to deep learning. The work is divided into two parts. The first part focuses on various aspects of neural ODEs, while the second part investigates the adoption of model reduction algorithms for use in standard artificial neural networks.

The second part is devoted to application of model order reduction to standard artificial neural networks (Chapters 5 and 6).

Contributions, Novelty, Impact

In Chapter 2, we present a novel interpolation-based method for approximating gradients in neural ODE models, which allows for more efficient training than the traditional reverse dynamic method (also known as the "adjoint method"). We demonstrate the effectiveness of our approach through a series of experiments involving classification, density estimation, and approximation tasks and provide a theoretical justification of our method using logarithmic norm formalism. This work was presented at the Neural Information Processing Systems (NeurIPS) conference in 2020 [36].

In Chapter 3, we examine the impact of various normalization techniques on the performance of neural ODEs. Through our analysis, we are able to achieve an accuracy of 93% in the CIFAR-10 classification task, which to the best of our knowledge, is the highest reported accuracy among neural ODEs that were tested for this problem. This work was presented at the ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations [66].

Chapter 4 investigates how the parametrization of a training ODE solver impacts the adversarial robustness of neural ODEs. We propose simple methods for improving the robustness of neural ODEs without incurring additional computational costs. This work has not yet been published but is available on arXiv [66].

In Chapter 5, we introduce a method for speeding up the inference process in deep neural networks through the use of reduced-order modeling techniques inspired by the analysis of dynamical systems. Our method, which is based on the maximum volume algorithm [116], demonstrates efficiency on pre-trained neural networks across a variety of datasets. We also show that it is possible to replace convolutional layers with fully-connected layers in a reduced dimention with minimal loss in accuracy. This work was published in the Computational Mathematics and Mathematical Physics Journal [65].

In Chapter 6, we explore the application of the active subspace method to deep learning. The active subspace method is a model reduction technique commonly used for differential equations, and we propose using it to analyze the internal structure and vulnerability of deep neural networks. Specifically, we use the active subspace method to measure the number of "active neurons" at each intermediate layer, which allows us to develop a more compact network (referred to as ASNet) with significantly fewer model parameters. We also propose using the active subspace method to analyze the vulnerability of neural networks by identifying universal adversarial attack vectors that can misclassify a dataset with high probability. Our experiments on the CIFAR-10 dataset show that ASNet can achieve 23.98x parameter reduction and 7.30x flops reduction, and the universal active subspace attack vector can achieve around 20% higher attack ratio compared to existing approaches in our numerical experiments. This work was published in the SIAM Journal of Mathematics for Data Science [35].

Chapter 1

Preliminaries

In this chapter, we provide an overview of the problems considered in this work, as well as an intuition of the standard algorithms and models that will be used in the following chapters.

In Section 1.1, we briefly formulate the considered machine learning problems and describe the loss functions.

In Section 1.2, we give all the required information about the neural ordinary differential equations (Neural ODEs) model. Neural ODEs models are the object of research in the first part of this work.

In Section 1.3, we describe the Discrete Empirical Interpolation Method for ODEs and show the connection to the Maximum Volume algorithm.

In Section 1.4, we overview the Active Subspaces method and show its applications to systems of ODEs.

1.1 Considered Machine Learning Problems

Classification

In multiclass classification, we are given a set of possible data points or samples $\mathcal{X} \subseteq \mathbb{R}^d$ and a finite set of classes $\mathcal{Y} = \{c_1, \ldots, c_k\}$. Each data point $\boldsymbol{x} \in \mathcal{X}$ corresponds to a single class $y \in \mathcal{Y}$, meaning that there exists an unknown function $f : \mathcal{X} \to \mathcal{Y}$.

The set $\mathcal{X}$ can be infinite, and typically we can not observe all its elements. Instead, we are given training samples and their corresponding classes as finite subsets $X \triangleq \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and $Y \triangleq \{y_1, \ldots, y_n\}$. Our task is to obtain (or train) a function $a : \mathcal{X} \to \mathcal{Y}$ that can predict the unknown mapping f even for $\boldsymbol{x} \notin X$. This function is called a classifier. Typically, we select a family of functions that are defined up to a set of parameters $\boldsymbol{\theta}$, and the problem of training a classifier is usually reduced to selecting a proper value of $\boldsymbol{\theta}$ by minimizing a so-called loss function with respect to $\boldsymbol{\theta}$.

There are many ways to measure the quality of the obtained function a. The simplest one is accuracy which is the average number of matches with the correct class for a special validation set that is not used during the training. Usually, the classifier outputs not a single class but an ordering with respect to its internal estimation of probability. In this case, we can define the top-m accuracy as the average number of matches with the correct class in the top-m predicted classes.

A common loss function for multiclass classification problem is the cross-entropy loss (also known as the negative log-likelihood loss) that for a single object $(\boldsymbol{x}, y)$ is defined as follows

$$L(\boldsymbol{\theta}, \boldsymbol{x}, y) \triangleq -\sum_{c \in \mathcal{Y}} [c = y] \log p_c(\boldsymbol{x}), \tag{1.1}$$

where squared brackets are the indicator function and $p_c(\boldsymbol{x})$ is the estimated probability that $\boldsymbol{x}$ belongs to the class c.

Density Estimation

In density estimation, we are given a set of samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ drawn from an unknown probability density function p. The goal is to create a model that can approximate the density function $p(\boldsymbol{x})$ for any unknown sample.

One common approach to training a parametric model for density estimation is to maximize the log-likelihood of the model with respect to the model parameters. The log-likelihood can also be used as a measure of the quality of the model.

For image datasets, a commonly used metric is bits per pixel (also known as bits/dim). This metric is calculated by computing the negative log-likelihood of the model using the binary logarithm, and then dividing it by the number of pixels in the image. For example, if the images in the dataset are tensors of shape $3 \times 32 \times 32$, the denominator for this metric would be 3072.

Universal Adversarial Attacks

Universal adversarial attacks refer to a type of adversarial attack on neural networks where a single adversarial perturbation can be applied to multiple input samples to cause the neural network to misclassify them. These attacks are called "universal" because they can be applied to a wide range of input samples rather than being specific to a single sample.

To understand the problem of universal adversarial attacks construction, it is first important to understand the concept of adversarial attacks in general. Adversarial attacks are a type of attack on machine learning models, including neural networks, where an attacker tries to cause the model to make incorrect predictions by making small, imperceptible changes to the input data. These changes are designed to be difficult for a human to detect but are still able to fool the model into making an incorrect prediction.

The problem with universal adversarial attacks is that they can be difficult to defend against since they can be applied to a wide range of input samples and do not require the attacker to carefully craft the perturbation for each individual sample. This makes it difficult for a model to be robust against these attacks since it is not possible to anticipate and defend against all possible perturbations. As a result, universal adversarial attacks can be a serious problem for neural networks, as they can cause the model to make incorrect predictions and potentially compromise the integrity of the system.

One of the applications of universal adversarial attacks is generating images for CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) to prevent artificial classifiers from recognizing the right answers.

1.2 Neural Ordinary Differential Equations

Neural ordinary differential equations (Neural ODEs) is a family of neural network architectures that contain the following system of ordinary differential equations as a building block

$$\begin{cases} \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}), \quad t \in [t_0, t_1] \\ \boldsymbol{z}(t_0) = \boldsymbol{z}_0. \end{cases} \tag{1.2}$$

We will call this block an *ODE block*. The right-hand side f is typically parametrized by a neural network, and parameters $\boldsymbol{\theta}$ are the weights of this neural network. The input to an ODE block is treated as the initial conditions of an ODE, and the output is usually the solution of this system of ODEs at the final time t_1.

Neural ODEs s

Runge–Kutta Methods and Adaptive Solvers

The forward pass through the ODE block is performed using numerical integrators of a system of ODEs. Here we give a brief overview of the numerical ODE integrators following the book by Hairer and Wanner [67].

Almost all numerical methods for solving ODEs can be viewed as a special case of the Runge–Kutta methods. Every Runge–Kutta method is defined by the number of stages s and parameters $a_{21}, a_{31}, a_{32}, \ldots, a_{s1}, a_{s2}, \ldots, a_{s,s-1}$, $b_1, \ldots, b_s, c_2, \ldots, c_s$. One step of the s-stage Runge–Kutta method for solving problem (1.2) is defined as

$$\begin{aligned} \boldsymbol{k}_1 &= f(\boldsymbol{z}_0, t_0, \boldsymbol{\theta}) \\ \boldsymbol{k}_2 &= f(\boldsymbol{z}_0 + h a_{21} \boldsymbol{k}_1, t_0 + c_2 h, \boldsymbol{\theta}) \\ \boldsymbol{k}_3 &= f(\boldsymbol{z}_0 + h(a_{31}\boldsymbol{k}_1 + a_{32}\boldsymbol{k}_2), t_0 + c_3 h, \boldsymbol{\theta}) \\ &\ldots \\ \boldsymbol{k}_s &= f(\boldsymbol{z}_0 + h(a_{s1}\boldsymbol{k}_1 + \ldots + a_{s,s-1}\boldsymbol{k}_{s-1}), t_0 + c_s h, \boldsymbol{\theta}) \\ \boldsymbol{z}_{t_0+h} &= \boldsymbol{z}_0 + h(b_1\boldsymbol{k}_1 + \ldots + b_s\boldsymbol{k}_s), \end{aligned}$$

where h is the step size.

A convenient way to visualize the coefficients of the Runge–Kutta method is a so-called Butcher tableau

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & & & \dots & & \\ c_s & a_{s,1} & a_{s,2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s. \end{array} \tag{1.3}$$

Thus, there is a one-by-one correspondence between the Runge–Kutta method and its Butcher tableau.

An important characteristic of a Runge–Kutta method is its order. A Runge–Kutta method is said to have the order p if for sufficiently smooth initial value problems problems (1.2) holds

$$\|z(t_0 + h) - z_{t_0+h}\| \leq Kh^{p+1}.$$

Methods are usually selected to have a high order p.

1.2.1 Adaptive Runge–Kutta Solvers

Each step of the Runge–Kutta method requires the selection of the step size h. It can be defined as a constant number, but two undesired situations are possible: lack of convergence due to the huge value of step and slow convergence due to an unreasonably small value of step. In order to manage these issues, automatic step selection procedures were developed.

The intuition behind the most popular adaptive Runge–Kutta methods is rather simple. Let us take two methods of different orders, say, $p = 5$ and $\widehat{p} = 4$. To avoid extra right-hand side function computations, two methods with common $a_{i,j}$ and c_i coefficients and different b_i coefficients are usually selected. We pick an initial step size h and want outputs z_h and $\widehat{z}_h$ of different solvers for the initial-value problem 1.2 to be approximately the same. The desired similarity of two outputs is controlled by two hyperparameters: absolute tolerance (**atol**) and relative tolerance (**rtol**). Formally, we want $\|z_h - \widehat{z}_h\|$ to be less than $\mathbf{sc} \triangleq \mathbf{atol} + \max\{\|z_h\|, \|\widehat{z}_h\|\} \cdot \mathbf{rtol}$ for a given norm. In the case of ℓ_2-norm, we can define the error term as

$$\text{err} \triangleq \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{z_{h,i} - \widehat{z}_{h,i}}{\mathbf{sc}_i}\right)^2}. \tag{1.4}$$

Then, the step is accepted if $\text{err} \leq 1$. It can be shown [67] that $\text{err} \approx Ch^{q+1}$, where $q \triangleq \min(p, \widehat{p})$. Since we want the error term to be equal to 1, the optimal step can be computed as

$$h_{\text{opt}} = h \cdot (1/\text{err})^{1/(q+1)}. \tag{1.5}$$

In practical implementations of adaptive solvers, additional simple heuristics are introduced to keep the step from shrinking too much or increasing too much.

One of the most common solvers for Neural ODEs is DOPRI5 which consists of two embedded Runge–Kutta methods of orders 5 and 4.

An important part of the Runge–Kutta methods is a so-called dense output [83, 156, 157], i.e., an ability to interpolate the solution at any point τ without a lot of additional computations as

$$z(\tau) = z_0 + h \sum_{i=1}^{s^*} b_i(\tau) \boldsymbol{k}_i, \tag{1.6}$$

where $\boldsymbol{k}_i$ are computed as in the Runge–Kutta formulas and s^* is a number of stages.

Backward pass through ODE blocks

Neural ODEs, like classical artificial neural networks, are trained via gradient methods. Thus, an important question is how to compute gradients of the loss function with respect to the parameters of Neural ODEs. In Chapter 2, we propose a stable algorithm for the computation of the gradients. However, in this subsection, we briefly describe classical approaches to this problem and derive formulas for the so-called adjoint method.

Obviously, a standard backpropagation can be applied to a numerical ODE solver. However, we can run out of memory since there is a need to store all intermediate values.

In order to differentiate ODEs using a limited amount of memory, there exists the so-called adjoint method. To derive formulas of this method, we will use the calculus of variations and Lagrange multipliers. Nevertheless, Chen et al. [22] derived formulas for this method in a slightly different way.

We have to minimize $L(z(t_1), \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ with constraints

$$\frac{\mathrm{d}z}{\mathrm{d}t} = f\left(z\left(t\right), \boldsymbol{\theta}\right), \quad z(t_0) = z_0. \tag{1.7}$$

This problem can be viewed as a constrained optimization problem with an infinite number of equality constraints. In this case, we introduce Lagrange $\boldsymbol{a}(t)$ and b [5, 13] and write the Lagrange function

$$\mathcal{L}(z(t), \boldsymbol{\theta}, \boldsymbol{a}(t)) = L(z(t_1), \boldsymbol{\theta}) + \int_{t_0}^{t_1} \boldsymbol{a}(t)^\top \left(\frac{\mathrm{d}z}{\mathrm{d}t} - f(z(t), \boldsymbol{\theta}) \right) \mathrm{d}t + b \cdot (z(t_0) - z_0). \tag{1.8}$$

s in discrete cases imply that all derivatives of the Lagrange function should be equal to zero. In the case of functions, we have to consider variational derivatives. Recall that the variational derivative of a function $J(y)$ is

$$\delta J(y, h) \triangleq \lim_{\varepsilon \to 0} \frac{J[y + \varepsilon h] - J[y]}{\varepsilon} = \frac{\partial}{\partial \varepsilon} J[y + \varepsilon h]\big|_{\varepsilon = 0}, \tag{1.9}$$

where h is a feasible variation.

Let us take the derivative of the Lagrange function with respect to $\boldsymbol{\theta}$. We can consider a standard derivative since $\boldsymbol{\theta}$ is not a function.

$$\frac{\partial \mathcal{L}(\boldsymbol{z}(t), \boldsymbol{\theta}, \boldsymbol{a}(t))}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \boldsymbol{\theta}}(\boldsymbol{z}(t_1), \boldsymbol{\theta}) - \int_{t_0}^{t_1} \boldsymbol{a}(t)^\top \frac{\partial f}{\partial \boldsymbol{\theta}}(\boldsymbol{z}(t), t, \boldsymbol{\theta})\, \mathrm{d}t = \mathbf{0}. \tag{1.10}$$

Condition (1.10) can be rewritten as

$$\frac{\partial L}{\partial \boldsymbol{\theta}}(\boldsymbol{z}(t_1), \boldsymbol{\theta}) = \mathbf{0} + \int_{t_1}^{t_0} \left(-\boldsymbol{a}(t)^\top \frac{\partial f}{\partial \boldsymbol{\theta}}(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \right) \mathrm{d}t. \tag{1.11}$$

This is the integral form of the solution of the following system of ODEs that are solved backward-in-time from t_1 to t_0

$$\begin{cases} \dfrac{\mathrm{d}}{\mathrm{d}t}\left(\dfrac{\partial L}{\partial \boldsymbol{\theta}}\right) = -\boldsymbol{a}(t)^\top \dfrac{\partial f}{\partial \boldsymbol{\theta}}(\boldsymbol{z}(t), \boldsymbol{\theta}), \\ \left.\dfrac{\partial L}{\partial \boldsymbol{\theta}}\right|_{t=t_1} = \mathbf{0}. \end{cases} \tag{1.12}$$

This system should be solved in order to compute the gradient of L with respect to $\boldsymbol{\theta}$, but it depends on $\boldsymbol{a}(t)$ and $\boldsymbol{z}(t)$.

To obtain the system for $\boldsymbol{a}(t)$, we take the functional derivative of the Lagrange function (1.8) with respect to $\boldsymbol{z}(t)$:

$$\begin{aligned} \delta_z \mathcal{L}(\boldsymbol{z}, \boldsymbol{h}) = \frac{\partial L}{\partial \boldsymbol{z}}(\boldsymbol{z}(t_1), \boldsymbol{\theta}) \cdot \boldsymbol{h}(t_1) + \\ + \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \int_{t_0}^{t_1} \boldsymbol{a}(t)^\top \left(\varepsilon \frac{\mathrm{d}\boldsymbol{h}(t)}{\mathrm{d}t} - \frac{\partial f}{\partial \boldsymbol{z}}(\boldsymbol{z}(t), \boldsymbol{\theta}) \cdot \varepsilon \boldsymbol{h}(t) + o(\varepsilon) \right) \mathrm{d}t \end{aligned} \tag{1.13}$$

Here $\boldsymbol{h}(t_0) = 0$, since $\boldsymbol{z}(t_0) + \varepsilon \boldsymbol{h}(t_0)$ should satisfy the initial conditions.

We apply integration by parts

$$\begin{aligned} \int_{t_0}^{t_1} \boldsymbol{a}(t)^\top \frac{\mathrm{d}\boldsymbol{h}(t)}{\mathrm{d}t} \mathrm{d}t = \left[\boldsymbol{a}(t)^\top \boldsymbol{h}(t) \right] \Big|_{t_0}^{t_1} - \int_{t_0}^{t_1} \frac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} \boldsymbol{h}(t)\, \mathrm{d}t = \\ = -\boldsymbol{a}(t_1)^\top \boldsymbol{h}(t_1) - \int_{t_0}^{t_1} \frac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} \boldsymbol{h}(t)\, \mathrm{d}t \end{aligned} \tag{1.14}$$

and get

$$\begin{aligned} \delta_z \mathcal{L}(\boldsymbol{z}, \boldsymbol{h}) = \left(\frac{\partial L}{\partial \boldsymbol{z}}(\boldsymbol{z}(t_1)) - \boldsymbol{a}(t_1) \right) \cdot \boldsymbol{h}(t_1) - \\ - \int_{t_0}^{t_1} \left\{ \frac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} + \boldsymbol{a}(t)^\top \frac{\partial f}{\partial \boldsymbol{z}}(\boldsymbol{z}(t), \boldsymbol{\theta}) \right\} \boldsymbol{h}(t) \mathrm{d}t = 0. \end{aligned} \tag{1.15}$$

Due to the main lemma of variational calculus,

$$\begin{cases} \dfrac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} = -\boldsymbol{a}(t)^\top \dfrac{\partial f}{\partial \boldsymbol{z}}\left(\boldsymbol{z}(t), \boldsymbol{\theta}\right) \\ \boldsymbol{a}(t_1) = \dfrac{\partial L}{\partial \boldsymbol{z}}\left(\boldsymbol{z}\left(t_1\right)\right). \end{cases} \tag{1.16}$$

Finally, we can take the functional derivative with respect to $\boldsymbol{a}(t)$ and obviously get

$$\begin{cases} \dfrac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \\ \boldsymbol{z}(t_1) = \boldsymbol{z}_1, \end{cases} \tag{1.17}$$

where $\boldsymbol{z}_1$ is $\boldsymbol{z}(t_1)$ saved after the forward pass.

To sum up, in the adjoint method, we have to solve the following system of ODEs

$$\begin{cases} \dfrac{\mathrm{d}}{\mathrm{d}t}\left(\dfrac{\partial L}{\partial \boldsymbol{\theta}}\right) = -\boldsymbol{a}(t)^\top \dfrac{\partial f}{\partial \boldsymbol{\theta}}\left(\boldsymbol{z}\left(t\right), \boldsymbol{\theta}\right) \\ \dfrac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \\ \dfrac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} = -\boldsymbol{a}(t)^\top \dfrac{\partial f}{\partial \boldsymbol{z}}\left(\boldsymbol{z}(t), \boldsymbol{\theta}\right) \\ \dfrac{\partial L}{\partial \boldsymbol{\theta}}\Big|_{t=t_1} = \boldsymbol{0} \\ \boldsymbol{z}(t_1) = \boldsymbol{z}_1 \\ \boldsymbol{a}(t_1) = \dfrac{\partial L}{\partial \boldsymbol{z}}\left(\boldsymbol{z}\left(t_1\right)\right) \end{cases} \tag{1.18}$$

backward-in-time from t_1 to t_0. The only required value is $\boldsymbol{z}_1$, which is saved after the forward pass. As a result, we obtain the gradient of the loss function with respect to parameters, and this value can be used with stochastic gradient optimization algorithms. In Chapter 2, we describe the disadvantages of this algorithm and give a receipt on how to mitigate the issues.

Continuous Normalizing Flows for Density Estimation

Discrete normalizing flows [98, 139] are popular generative models. In order to define discrete normalizing flows, we have to select a family of invertible functions and a probability distribution (say, standard Gaussian). Then, we try to generate real data from noise by applying functions from the selected family. The main workhorse of discrete normalizing flows is the change of variables theorem: if $\boldsymbol{x} = f(\boldsymbol{z})$, then

$$\log p(\boldsymbol{x}) = \log p(\boldsymbol{z}) + \log \left| \det \frac{\partial f^{-1}}{\partial \boldsymbol{x}} \right|. \tag{1.19}$$

The need to compute the determinant of the inverse transformations is a serious constraint since, in the general case, it is intractable. That is why, in discrete normalizing flows, families of functions with tractable Jacobians are introduced.

Chen et al. [22] introduced continuous normalizing flows. In continuous normalizing flows, the data is transformed via the system of ODEs 2.1. It can be shown that under mild conditions, the transformation of the probability density function is also guided by the system of ODEs

$$\frac{\mathrm{d}\log p(\boldsymbol{z}(t))}{\mathrm{d}t} = -\mathrm{tr}\left(\frac{\partial f}{\partial \boldsymbol{z}(t)}\right), \quad \log p(z(t_0)) = \log p_{z_0}(\boldsymbol{z}_0), \tag{1.20}$$

where p_{z_0} is a base distribution from which the first sample is drawn. To reduce the computational time, Grathwohl et al. [61] proposed to use the Hutchinson estimation [87] of the trace instead of its exact computation.

1.3 Discrete Empirical Interpolation Method

Let us consider a system of ODEs

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{y}(t) = \boldsymbol{A}\boldsymbol{y}(t) + \boldsymbol{F}(\boldsymbol{y}(t)) \tag{1.21}$$

with any initial condition, where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is a matrix and $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$ is an elementwise non-linear function. Such systems typically come from the discretization of several classes of PDEs. Suppose we want to reduce the computational complexity and the memory footprint but obtain approximately the same output. It can be done by reducing the system dimensionality.

Model reduction methods usually search for a linear subspace of order $k \ll n$ that approximates the original system. The subspace is determined by a matrix $\boldsymbol{V}_k \in \mathbb{R}^{n\times k}$, and the new system looks as follows

$$\frac{\mathrm{d}}{\mathrm{d}t}\widetilde{\boldsymbol{y}}(t) = \boldsymbol{V}_k^\top \boldsymbol{A}\boldsymbol{V}_k\widetilde{\boldsymbol{y}}(t) + \boldsymbol{V}_k^\top \boldsymbol{F}(\boldsymbol{V}_k\widetilde{\boldsymbol{y}}(t)). \tag{1.22}$$

The information about the system is usually stored as a set of snapshots, i.e., inputs, intermediate points, and outputs.

The main difference between the different model reduction techniques is the choice of subspace. In the proper orthogonal decomposition via Galerkin projection (POD-Galerkin) [142, 100], the subspace is computed via the singular value decomposition (SVD) of the snapshot matrices. This method is optimal with respect to the approximation error. However, POD-Galerkin has a high complexity since the non-linear function $\boldsymbol{F}$ still should be computed for n-dimensional vectors, although the linear term can be precomputed.

Another popular model reduction method is the Discrete Empirical Interpolation Method (DEIM) [21]. The key difference of the DEIM algorithm is the use of sparse sampling to identify a set of "interpolation points" that are used to approximate the high-dimensional function. In sparse sampling, a subset of the data points is selected in a way that captures the key features of the function while minimizing the number of points needed. This can be done using various techniques, such as greedy algorithms or random sampling. (In Chapter 5, we use the maximum volume algorithm for this task [60, 116].)

DEIM is particularly well-suited for systems with non-linear functions $\boldsymbol{F}$ that are expensive to compute, as it allows for efficient evaluation of $\boldsymbol{F}$ at the selected indices.

In summary, both POD-Galerkin and DEIM are methods for constructing reduced-order models of systems of ODEs, but they differ in the way they determine the subspace that approximates the original system. POD-Galerkin is optimal with respect to the approximation error, but it has a high computational complexity, while DEIM is more efficient but may not be as accurate.

1.4 Active Subspace Method

The active subspace method is a technique used in optimization and sensitivity analysis to identify the most important directions, or "subspaces," in the input space of a function. It can be used to reduce the dimensionality of the input space, making optimization and sensitivity analysis more efficient.

The active subspace method is based on the idea that, for many functions, the most important directions in the input space are those that have the greatest effect on the output of the function. These directions are called the "active subspaces," and they can be identified using the gradient of the function with respect to the inputs.

The first paper describing the active subspace method Constantine, Dow, and Wang [27] showed simulations for elliptic partial differential equations. Later on, this method was widely used to analyze systems governed by differential equations.

For example, in fluid dynamics, the Navier-Stokes equations are a set of differential equations that describe the motion of a fluid. These equations can be very complex, especially in three-dimensional space, but the active subspaces method can be used to identify the most important directions in the space of solutions, which can help us understand the behavior of the fluid flow [37]. The active subspaces were successfully used for applied problems such as HIV modeling [109], lithium-ion battery modeling [26], and so on.

Overall, the active subspaces method is a powerful tool for analyzing and interpreting systems governed by differential equations, and it can be used in many different fields of science and engineering. A detailed and formal mathematical description of the active subspace method is given in Chapter 6.

Chapter 2

Acceleration of Gradients Propagation in Neural ODEs

2.1 Introduction

In this chapter, we propose a novel method to train neural ordinary differential equations (neural ODEs) [22]. This method performs stable and memory-efficient backpropagation through the solution of initial value problems (IVP). We use the term neural ODEs for all neural networks with so-called ODE blocks. An ODE block is a continuous analog of a residual neural network [73] that can be considered as Euler discretization of ordinary differential equations.

As we already mentioned in the previous chapter, ODE block is a neural network layer that takes the activations $\boldsymbol{z}_0$ from the previous layer as input, solves the initial value problem (IVP) described as

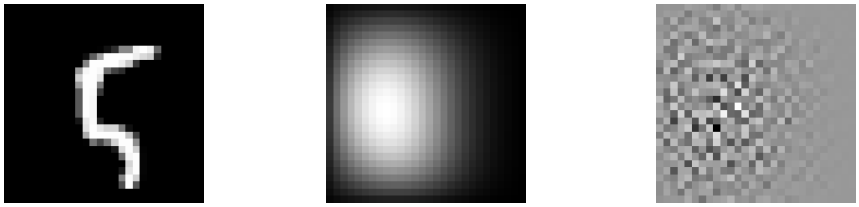
$$\begin{cases} \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}), \quad t \in [t_0, t_1] \\ \boldsymbol{z}(t_0) = \boldsymbol{z}_0, \end{cases} \tag{2.1}$$

and solved by any ODE solver. Note, the right-hand side in IVP (2.1) depends on set of parameters $\boldsymbol{\theta}$ which is gradually updated during training. Therefore, during the backward pass in neural ODEs, the loss function L, which depends on the solution of the IVP, should be differentiated with respect to parameter $\boldsymbol{\theta}$. A direct application of backpropagation to ODE solvers require a huge memory, since for every time step τ_k, the output $\boldsymbol{z}(\tau_k)$ must be stored as a part of the computational graph.

However, the approach based on the *adjoint method* [132, 114] helps to propagate gradients through the initial value problem with a relatively small memory footprint. Originally adjoint method has been actively used in mathematical modeling, for example, in seismograph and climate studies [44, 69]. It was used to investigate the sensitivity of the model output with respect to the input. In the context of training neural ODEs, the adjoint method is modified to combine it with the standard backpropagation [22]. We refer to this modified adjoint method as *reverse dynamic method* (RDM). This method yields solving the augmented initial value problem backward-in-time. We call this IVP as the *adjoint IVP*. One of the components of the adjoint IVP is IVP that defines $\boldsymbol{z}(t), t \in [0, 1]$. Therefore, the numerical solving of the adjoint IVP

does not require storing intermediate activations $z(\tau_k)$ during the forward pass. As a result, the memory footprint becomes smaller.

However, Gholami et al. [55] showed that the RDM might lead to catastrophic numerical instabilities. We checked this fact by ourselves and got the same results.



z_0 z_1 $\widehat{z}_0$

FIGURE 2.1: Example of potential instability of RDM as described in [55]. We took a single ResNet-block with random weights as a right-hand side and solved the initial value problem 2.1 with an initial condition z_0 (left) and obtained the image z_1 (middle). After that, we integrated the same initial value problem backward-in-time and obtained $\widehat{z}_0$ (right). Images on the left and on the right should coincide in order to perform an accurate backward pass, but they obviously do not.

To address this issue, the authors introduce a method called ANODE. This method exploits *checkpointing* idea [154], i.e., propose to store checkpoints $z(\tau_k)$ at intermediate selected time points τ_k during the forward pass. As result of this, in the backward pass, ANODE performs additional ODE solver steps forward-in-time in each interval between sequential checkpoints. The intermediate activations are stored to compute the target gradient. The main disadvantage of ANODE is that it requires intermediate activations storage and needs to perform additional ODE solver steps.

To address the instability of the reverse dynamic method and limitations of ANODE, we propose *interpolated reverse dynamic method* (IRDM), which is described in detail in Section 2.3. This method is based on a smooth function interpolation technique to approximate $z(t)$ and exclude the IVP that defines $z(t)$ from the adjoint IVP. Thus, we do not reverse IVP (2.1) and avoid the instability problem of the reverse dynamic method. Under mild conditions on the right-hand side $f(z(t), t, \boldsymbol{\theta})$, function $z(t)$ is continuously differentiable as a solution of IVP, i.e. $z(t) \in C^1[t_0, t_1]$. Therefore, it can be approximated with the *barycentric Lagrange interpolation* (BLI) [10] on a Chebyshev grid. This technique is widely used for interpolation problems [10]. To construct such approximation, one has to store activations $z(t)$ in the point from the Chebyshev grid during the forward pass. These activations can be computed with DOPRI5 adaptive ODE solver without additional right-hand side evaluations [40]. After that, in the backward pass, stored activations are used to approximate $z(t)$ during the adjoint IVP solving. The main requirement for our method to work correctly and efficiently is that $z(t)$ can be approximated by BLI with sufficient accuracy. This can only be verified experimentally. However, the accuracy of such approximation is inherently related to the smoothness of

the solution, which is also one of the main motivations behind using neural ODEs.

Our main contributions are the following.

- We propose the *interpolated reverse dynamic method* to train neural ODEs. This method uses approximated activations $z(t)$ in the backward pass and reduces the dimension of the initial value problem, that is used to compute the gradient. Thus, the training becomes faster.
- We present the error bound for the gradient norm under small perturbation of the activations $z(t)$ induced by using interpolated values.
- We have evaluated our approach on density estimation, inference approximation, and classification tasks and showed its effectiveness in terms of test loss-training time trade-off compared to the reverse dynamic method.

2.2 Related Work

Neural ODEs [22] is a model inspired by the connection between neural networks and dynamical systems [110, 20, 146, 136]. Neural ODEs and its modifications were used for various different applications [61, 144, 66, 182, 41]. Nguyen et al. [123] emphasized the importance of using adaptive solvers and introduce a procedure to learn their tolerances. Quaglino et al. [133] proposed to use of spectral element methods, where the dynamics are expressed as a truncated series of Legendre polynomials. Similar ways of using interpolation in the adjoint method are implemented in SUNDIALS [78].

The proposed method relies on the ability of Runge-Kutta (RK) methods to evaluate the trajectory in intermediate points with Hermite polynomial interpolation. We use this feature of RK methods to evaluate activations in the Chebyshev grid points. The works by L. F. Shampine [156, 157] study the error induced by this approach to evaluate activations in intermediate points. In addition, the stiffness of ODE is an important concept [161, 169] for stable and fast training of neural ODEs.

2.3 Interpolated Reverse Dynamic Method

Deep learning problems are usually solved by minimizing a loss function L with respect to model parameters using gradient-based methods. To compute the gradient $\frac{\partial L}{\partial \theta}$ without saving computational graph from the forward pass, the *adjoint method* can be used [58, 131].

Adjoint method. The detailed derivation of the adjoined method is given in Section 1.2.1. However, we recall the basics of this approach. The main idea of the adjont method is to derive gradients of the loss function L from the first-order optimality conditions (FOOC) for a constrained optimization

problem. In our case, the optimization problem is formulated as the loss minimization with ODE constraint in the form of (2.1).

To construct the corresponding Lagrangian, the adjoint variable $\boldsymbol{a}(t)$ is introduced

$$\mathcal{L}(\boldsymbol{z}(t), \boldsymbol{\theta}, \boldsymbol{a}(t)) = L(\boldsymbol{z}(t_1), \boldsymbol{\theta}) + \int_{t_0}^{t_1} \boldsymbol{a}(t) \left(\frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} - f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \right) dt,$$

and the FOOC can be written in the following form

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{a}(t)} = \boldsymbol{0} \rightarrow \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} - f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) = \boldsymbol{0} \tag{2.2}$$

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{z}(t)} = \boldsymbol{0} \rightarrow \begin{cases} \frac{\mathrm{d}\boldsymbol{a}}{\mathrm{d}t} = -\boldsymbol{a}(t)^\top \frac{\partial f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{z}} \\ \boldsymbol{a}(t_1) - \frac{\partial L}{\partial \boldsymbol{z}(t_1)} = \boldsymbol{0} \end{cases} \tag{2.3}$$

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{\theta}} = \boldsymbol{0} \rightarrow \frac{\partial L}{\partial \boldsymbol{\theta}} = \int_{t_0}^{t_1} \boldsymbol{a}(t)^\top \frac{\partial f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} dt. \tag{2.4}$$

Hence, the target gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$ can be computed in the following way: ODE (2.2) gives the activation dynamic $\boldsymbol{z}(t)$, ODE (2.3) gives the adjoint variable $\boldsymbol{a}(t)$ based on $\boldsymbol{z}(t)$ and finally the target gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$ is computed with the integral in (2.4). The adjoint method assumes that activations $\boldsymbol{z}(t_0) = \boldsymbol{z}_0$ are known in the backward pass. Thus, IVP (2.1) is solved forward-in-time, IVP (2.3) is solved backward-in-time and integral (2.4) is computed based on the derived $\boldsymbol{a}(t)$ and $\boldsymbol{z}(t)$. The adjoint method requires storing gradients $\frac{\partial f}{\partial \boldsymbol{\theta}}$ and $\frac{\partial f}{\partial \boldsymbol{z}}$ in intermediate activations $\boldsymbol{z}(t), t \in [t_0, t_1]$. Therefore, to reduce its memory consumption, the checkpointing idea is used.

Checkpointing in the adjoint method. ANODE method [55] exploits checkpointing idea to get the target gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$. This method stores some intermediate activations $\boldsymbol{z}(t)$ in the forward pass. These activations are called checkpoints. In the backward pass, ANODE considers intervals between sequential checkpoints from the right side to the left side. In every interval, ODE (2.2) with an initial condition equal to the checkpoint on the left is solved forward-in-time, IVP (2.3) is solved backward-in-time and the integral is updated. This approach is illustrated in Figure 2.2d. This approach still requires additional memory to store checkpoints and gradients $\frac{\partial f}{\partial \boldsymbol{\theta}}$. Also, it solves ODE (2.2) with multiple initial conditions equal to checkpoints. These drawbacks are fixed in reverse dynamic method [22].

Reverse dynamic method. This method is used in [22], where the neural ODE model was proposed, under the name "adjoint method". The reverse dynamic method assumes that activations $\boldsymbol{z}(t_1) = \boldsymbol{z}_1$ are known and do not store any checkpoints during the backward pass. Therefore, in the backward

pass, the following IVP is solved backward-in-time and defines activaions:

$$\begin{cases} \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) \\ \boldsymbol{z}(t_1) = \boldsymbol{z}_1, \end{cases} \tag{2.5}$$

IVP (2.3) is solved backward-in-time and integral (2.4) is computed as the solution of the following IVP:

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \boldsymbol{\theta}}\right) = -\boldsymbol{a}(t)^\top \frac{\partial f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\ \frac{\partial L}{\partial \boldsymbol{\theta}}(t_1) = \mathbf{0}. \end{cases} \tag{2.6}$$

Thus, IVPs (2.3),(2.5) and (2.6) can be composed in the augmented IVP that is solved backward-in-time. This method is illustrated in Figure 2.2a. The study [55] demonstrates that this method can be unstable due to the reverse IVP (2.1). To get the right trade-off between stability and memory consumption, we propose *interpolated reverse dynamic method* (IRDM).

Interpolated reverse dynamic method. In the proposed *interpolated reverse dynamic method* (IRDM), we suggest to eliminate (2.5) from the adjoint IVP. Instead of using IVP (2.5) to get activations $\boldsymbol{z}(t)$, the IRDM approximates them through the barycentric Lagrange interpolation (BLI) on a Chebyshev grid [10]. This method is summarized in Figure 2.2c. We urge readers not to confuse the Lagrange interpolation, which is mostly of theoretical interest, with the BLI, that is widely used in practice for polynomial interpolation [77].

Denote by $\hat{\boldsymbol{z}}(t)$ the interpolated activations with the BLI technique that are used in the backward pass. As described in [77], $\hat{\boldsymbol{z}}(t)$ can be computed with the following equation:

$$\hat{\boldsymbol{z}}(t) = \left(\sum_{n=0}^{N} \frac{w_n}{t - \tau_n} \hat{\boldsymbol{z}}_n\right) \Bigg/ \left(\sum_{n=0}^{N} \frac{w_n}{t - \tau_n}\right), \tag{2.7}$$

where the sequence $\{\tau_n\}_{n=0}^{N}$ form the Chebyshev grid, and $t_0 = \tau_0 < \tau_1 < \ldots < \tau_N = t_1$, $\hat{\boldsymbol{z}}_n \triangleq \boldsymbol{z}(\tau_n)$ are exact activations computed in the Chebyshev grid during the forward pass and stored to be used in the backward pass. To get these activations during the forward pass, we explore features of DOPRI5 adaptive solver [40] to compute activations in given time points (e.g., in Chebyshev grid) without additional right-hand side evaluations. Thus, we store $\boldsymbol{z}(\tau_n)$ and solve IVP (2.1) simultaneously. The weights w_n are computed as follows once for the entire training process

$$w_n = (-1)^n \sin\left(\frac{(2n+1)\pi}{2N+2}\right).$$

The computational complexity of computing $\hat{\boldsymbol{z}}(t)$, as well as additional memory usage, is $O(N)$. Since we approximate $\boldsymbol{z}(t)$, only (2.3) and (2.6) have to be solved backward-in-time during the backward pass. Thus, the dimension of the backward IVP is reduced by the size of the activations $\boldsymbol{z}(t)$.

From the theory of polynomial interpolation, it is known, that if $z(t)$ is analytic function, then the following bound on the BLI approximation error holds

$$\max_{t\in[0,1]} \|\hat{z}(t) - z(t)\|_\infty \leq \mathcal{O}(M^{-N}), \tag{2.8}$$

where $M > 1$ depends on the region where the activation dynamic $z(t)$ is analytic, more details see in [77, 47, 165]. The solution of IVP (2.1) is analytic if the right-hand side $f(z(t), t, \boldsymbol{\theta})$ is analytic [45].

2.4 Upper Bound on the Gradient Error Induced by Interpolated Activations

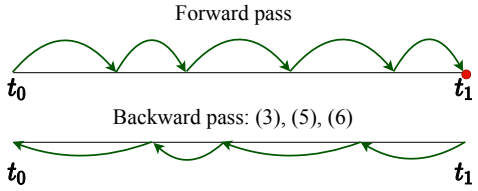
The proposed method excludes IVP that defines $z(t)$ from the adjoint IVP and uses approximation $\hat{z}(t)$ given by the barycentric Lagrange interpolation formula (2.7). Therefore, the dimension of the adjoint IVP is reduced, but the error in gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$ appears since the activations are not exact. In this section, we present the upper bound on the gradient error norm and the factors that affect the magnitude of this error. By the gradient error norm, we mean the norm of the difference between gradients computed with exact activations $z(t)$ and interpolated ones $\hat{z}(t)$.

According to (2.4) we have to estimate the following error norm, where the 2-norm is used

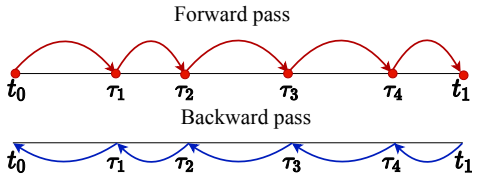
$$E = \left\| \int_{t_0}^{t_1} \left[\tilde{\boldsymbol{a}}(t)^\top \frac{\partial f(\tilde{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} - \boldsymbol{a}(t)^\top \frac{\partial f(z(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right] dt \right\|, \tag{2.9}$$

where $\tilde{\boldsymbol{a}}(t)$ and $\tilde{z}(t)$ are adjoint variables and activations obtained with interpolation technique and $\boldsymbol{a}(t)$ and $z(t)$ are exact ones. Now we derive the upper bound estimate of E and show what factors affect this upper bound. Introducing perturbations $\Delta z(t)$ and $\Delta \boldsymbol{a}(t)$ such that $\tilde{\boldsymbol{a}}(t) = \boldsymbol{a}(t) + \Delta \boldsymbol{a}(t)$ and $\tilde{z}(t) = z(t) + \Delta z(t)$ and using the first-order expansion of $\frac{\partial f}{\partial \boldsymbol{\theta}}$ at $z(t)$ we can re-write terms from (2.9) with perturbed adjoint variables and activations as follows

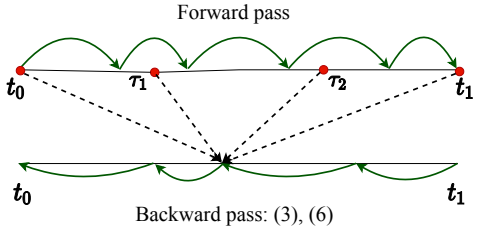
$$\tilde{\boldsymbol{a}}(t)^\top \frac{\partial f(\tilde{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = (\boldsymbol{a}(t) + \Delta \boldsymbol{a}(t))^\top \left(\frac{\partial f(z(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{\partial^2 f(z(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial z} \Delta z(t) + \right.$$
$$\left. + \mathcal{O}(\|\Delta z(t)\|^2) \right).$$



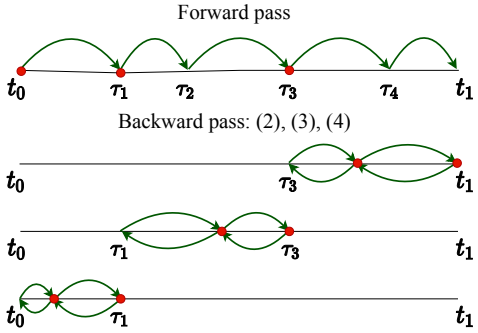
(A) Reverse dynamic method (RDM) [22], [61]



(B) Standard backpropagation



(C) Interpolated reverse dynamic method (IRDM)



(D) Checkpointing (ANODE)

FIGURE 2.2: Comparison of different schemes to make forward and backward passes through the ODE block. Red circles indicate that the activations are stored at these time points. Red arrows indicate that during ODE steps, the outputs of the intermediate layer are stored to propagate gradients. Green arrows correspond to the steps with ODE solvers. Blue arrows correspond to the steps with automatic differentiation through the stored computational graph. Activations in Chebyshev grid points (t_0, τ_1, τ_2 and t_1 in Figure 2.2c) are stored in the interpolation approach during the forward pass. Chebyshev grid points do not necessarily coincide with time steps of ODE solver, but activations in these points can be recovered from the computed activations with ODE solver. The stored activations are used to approximate activations in the backward pass. The dotted arrows in Figure 2.2c shows that activations in t_0, τ_1, τ_2 and t_1 are used to interpolate activations in the backward pass.

Substitution the above equality into (2.9) and applying standard inequalities lead to the following upper bound

$$E \leq \int_{t_0}^{t_1} \|\boldsymbol{a}(t)\| \|\Delta \boldsymbol{z}(t)\| \left\| \frac{\partial^2 f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial \boldsymbol{z}} \right\| dt + \int_{t_0}^{t_1} \|\boldsymbol{a}(t)\| \left\| \frac{\partial f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right\| dt + \int_{t_0}^{t_1} \|\Delta \boldsymbol{a}(t)\| \|\Delta \boldsymbol{z}(t)\| \left\| \frac{\partial^2 f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial \boldsymbol{z}} \right\| dt + M_z \int_{t_0}^{t_1} (\|\boldsymbol{a}(t)\| + \|\Delta \boldsymbol{a}(t)\|) \|\Delta \boldsymbol{z}(t)\|^2 dt, \tag{2.10}$$

where $M_z > 0$ is a constant hidden in big-O notation. In order to estimate the error norm, we have to analyze bounds for all terms in (2.10). The norm of activations perturbation $\|\Delta \boldsymbol{z}(t)\|$ is bounded according to the upper bound on the interpolation error (2.8). At the same time, the gradient $\frac{\partial f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ and the second partial derivative $\frac{\partial^2 f(\boldsymbol{z}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial \boldsymbol{z}}$ are not bounded a priori, so we need to consider them additionally. The remaining terms are $\|\boldsymbol{a}(t)\|$ and $\|\Delta \boldsymbol{a}(t)\|$ and to bound them we need the following Lemma [162].

Lemma 2.4.1 *([162], see Lemma 1)*
Let $\boldsymbol{x}(t)$ be a solution of the following non-autonomous linear system

$$\begin{cases} \dfrac{d\boldsymbol{x}}{dt} = \boldsymbol{x}(t)^\top \boldsymbol{A}(t) + \boldsymbol{b}(t), \\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0. \end{cases} \tag{2.11}$$

Then $\|\boldsymbol{x}(t)\| \leq \xi(t)$, where the scalar function ξ satisfies the IVP

$$\begin{cases} \dfrac{d\xi}{dt} = \mu[\boldsymbol{A}(t)]\xi + \|\boldsymbol{b}(t)\|, \\ \xi(t_0) = \|\boldsymbol{x}_0\|, \end{cases} \tag{2.12}$$

where $\mu[\boldsymbol{A}] \triangleq \lim\limits_{h \to 0+} \frac{\|\boldsymbol{I} + h\boldsymbol{A}\| - 1}{h}$ is a logarithmic norm of matrix $\boldsymbol{A}$ [160].

If the 2-norm is used in definition of the logarithmic norm, then

$$\mu[\boldsymbol{A}] = \lambda_{\max}\left(\frac{\boldsymbol{A} + \boldsymbol{A}^\top}{2}\right),$$

where $\lambda_{\max}(\boldsymbol{A})$ is the maximum eigenvalue of a matrix $\boldsymbol{A}$.

In our case, IVP (2.11) is equivalent to IVP (2.3). Therefore, this lemma helps to estimate $\|\boldsymbol{a}(t)\|$. In particular, $\|\boldsymbol{a}(t)\| \leq \xi(t)$, where $\xi(t)$ is a solution of the following IVP:

$$\begin{cases} \dfrac{d\xi}{dt} = \mu[\boldsymbol{J}(t)]\xi, \\ \xi(t_1) = \left\| \frac{\partial L}{\partial \boldsymbol{z}_1} \right\|, \end{cases} \tag{2.13}$$

where $\boldsymbol{J}(t) \triangleq \frac{\partial f(z(t),t,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$. Hence, the upper bound on the adjoint variable norm is written with the solution of IVP (2.13):

$$\|\boldsymbol{a}(t)\| \leq \xi(t_1) \exp\left(\int_{t_1}^{t} \mu[\boldsymbol{J}(\tau)] d\tau\right). \tag{2.14}$$

The upper bound for $\|\Delta \boldsymbol{a}(t)\|$ can be also obtained with Lemma 2.4.1. To derive this upper bound, we compose an auxiliary IVP that defines a dynamic of $\Delta \boldsymbol{a}(t)$. Consider the following IVPs corresponding to exact and perturbed activations:

$$\begin{cases} \frac{\mathrm{d}\boldsymbol{a}}{\mathrm{d}t} = \boldsymbol{a}(t)^\top \frac{\partial f(z(t),t,\boldsymbol{\theta})}{\partial z} \\ \boldsymbol{a}(t_1) = \frac{\partial L}{\partial z(t_1)} \end{cases} \quad \begin{cases} \frac{\mathrm{d}\tilde{\boldsymbol{a}}}{\mathrm{d}t} = \tilde{\boldsymbol{a}}(t)^\top \frac{\partial f(\tilde{z}(t),t,\boldsymbol{\theta})}{\partial z} \\ \tilde{\boldsymbol{a}}(t_1) = \frac{\partial L}{\partial z(t_1)}. \end{cases} \tag{2.15}$$

Subtracting one IVP from the other, we get the IVP that defines dynamic of $\Delta \boldsymbol{a}(t) = \tilde{\boldsymbol{a}}(t) - \boldsymbol{a}(t)$:

$$\begin{cases} \frac{\mathrm{d}\Delta \boldsymbol{a}(t)}{\mathrm{d}t} = \Delta \boldsymbol{a}(t)^\top \boldsymbol{J}(t) + \tilde{\boldsymbol{a}}(t)(\tilde{\boldsymbol{J}}(t) - \boldsymbol{J}(t)) \\ \Delta \boldsymbol{a}(t_1) = \boldsymbol{0}. \end{cases} \tag{2.16}$$

Note that IVP (2.16) satisfies assumption in Lemma 2.4.1. Therefore, the following estimate holds

$$\|\Delta \boldsymbol{a}(t)\| \leq \xi(t), \tag{2.17}$$

where $\xi(t)$ is a solution of the following IVP:

$$\begin{cases} \frac{\mathrm{d}\xi}{\mathrm{d}t} = \mu[\boldsymbol{J}(t)]\xi + \|\tilde{\boldsymbol{a}}(t)(\tilde{\boldsymbol{J}}(t) - \boldsymbol{J}(t))\|, \\ \xi(t_1) = \boldsymbol{0}. \end{cases} \tag{2.18}$$

The solution of IVP (2.18) is given by the following formula

$$\xi(t) = \phi(t) \int_{t_1}^{t} \phi^{-1}(\tau) \|\tilde{\boldsymbol{a}}(\tau)(\tilde{\boldsymbol{J}}(t) - \boldsymbol{J}(t))\| d\tau, \tag{2.19}$$

where $\phi(t) = \exp\left(\int_{t_1}^{t} \mu[\boldsymbol{J}(\tau)] d\tau\right)$ is a fundamental solution of IVP (2.19). Thus, we get the upper bounds for all terms in the inequality (2.10).

Thus, we can list the factors that affect the accuracy of gradient approximation with interpolated activations. These factors are constants that bound $\frac{\partial f(z(t),t,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ and $\frac{\partial^2 f(z(t),t,\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial z}$ for $t \in [t_0, t_1]$ (2.10), logarithmic norm $\mu[\boldsymbol{J}(t)]$ (2.14), (2.17), (2.19), and smoothness of $\boldsymbol{J}(t)$.

2.5 Numerical Experiments

In this section, firstly, we compare the proposed the IRDM with the RDM on the density estimation and variational inference tasks (for the RDM baselines, we use FFJORD [61] implementation). Secondly, we show the benefits of the

IRDM on the CIFAR10 classification task (RDM implementation is similar to [22]). We demonstrate that during training, the IRDM requires less computational time to achieve convergence and a smaller number of evaluations of the right-hand side function comparing to the baselines. The source code of the proposed method can be found at GitHub[1].

Also, as the number of Chebyshev grid points N is an important hyperparameter in our method, we study how it affects the gain in considered tasks. Our method is implemented on top of `torchdiffeq`[2] package. The default ODE solver in our experiments is the DOPRI5. The values of optimized hyperparameters are in the supplementary materials. Mostly we follow the strategies from [61] and [55]. Every separate experiment is conducted on a single NVIDIA Tesla V100 GPU with 16Gb of memory [178]. We conducted all experiments with three different fixed random seeds and reported the mean value. Experiments were tracked using the "Weights & Biases" library [11].

2.5.1 Experimental settings

To integrate ODE blocks in all experiments, we use the DOPRI5 ODE solver. We report mean test loss values and function evaluations values. These values are computed based on the ten runs with different fixed random seeds for toy datasets and three runs for other datasets.

2.5.1.1 Classification

We test considered methods of neural ODE model training in the CIFAR10 classification task. We consider a model with a single ODE block, which consists of two convolutional layers with 64 input and output channels, ReLU activations, and weight normalizations. A convolutional layer with three input channels and 64 output channels, a batch normalization layer, and ReLU activation precede the ODE block. We use only random crops and random flips for data augmentation.

The training is performed by SGD with momentum 0.9. The weight decay is equal to `1e-5`; the learning rate is fixed and equal to `5e-3`, batch size is 100. Initial absolute and relative tolerances are set to `1e-5`. After the $n^{\text{th}}150$ epoch, these tolerances are decreased by 10.

2.5.1.2 Density estimation

For the density estimation problem, we consider `miniboone` tabular dataset and four two-dimensional toy datasets. Data and preprocessing procedures are taken from `https://github.com/gpapamak/maf` and `https://github.com/rtqichen/ffjord`, respectively.

Instead of simple linear layers inside the ODE block, we use a so-called `concatsquash` linear layers, which are defined as follows

$$(\boldsymbol{W}\boldsymbol{z} + \boldsymbol{b}_1) \odot \sigma\left(t\boldsymbol{c} + \boldsymbol{b}_2\right) + t\boldsymbol{b}_3, \tag{2.20}$$

[1] `https://github.com/Daulbaev/IRDM`

[2] `https://github.com/rtqichen/torchdiffeq/`

where z are input activations, t stands for the time, $\boldsymbol{W}, \boldsymbol{c}, \boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3$ are trainable parameters, σ is a sigmoid activation, and $\odot$ is an element-wise product.

For the `miniboone` dataset, we train a model with 10 ODE blocks and softplus nonlinearity. It is trained by Adam optimizer with a fixed learning rate equal to `1e-3`. The batch size is equal to 10000. Absolute and relative tolerances are set to `1e-8` and `1e-7`, respectively. The training terminates if test loss does not decrease during 30 sequential epochs.

In experiments with toy datasets, a model with a single ODE block is used. This ODE block consists of three `concatsquash` linear layers of size 2×64, 64×64, and 64×2. To train the Neural ODEs model, SGD with momentum 0.9 and fixed learning rate `1e-3` is used. Absolute and relative tolerances are set to `1e-5`. The number of iterations in the training procedure is 10000, and 100 samples compose mini-batch.

2.5.1.3 VAE

For VAE experiments, we choose two datasets: `Caltech` and `Freyfaces`. Both datasets can be found in `https://github.com/riannevdberg/sylvester-flows`. All the experimental settings are exactly the same as in the FFJORD paper. The only difference is that IRDM is used to train models instead of RDM.

Improvement in stability of gradient computations. We perform experiments on the reconstruction trajectory of the dynamical system that collapses in zero. As a result, we observe that the reverse dynamic method (RDM) and our method (IRDM) solve this problem with approximately the same accuracy. However, the RDM requires at least 10 times more right-hand side evaluations to solve adjoint IVP in every iteration than the IRDM. We use RDM implementation from the `torchdiffeq` package. Thus, in such a toy problem the IRDM and the RDM compute similar gradients, but the IRDM computes them much faster. To illustrate the stability of the IRDM, we show the plot of test loss vs. training time in density estimation problem, see Figure 2.4a.

How gradient approximation depends on the tolerance in adaptive solver and size of the Chebyshev grid. In Section 2.4, we provide theoretical bounds on the gradient approximation and list the main factors that affect it. However, tolerance in the adaptive solver and number of nodes in the Chebyshev grid can also significantly affect the quality of gradient approximation. To illustrate this influence empirically, we consider the toy dynamical system in 2D with the right-hand side $\boldsymbol{A}\boldsymbol{y}^3$ and train neural ODE model to approximate trajectories of this dynamical system. We consider the range of tolerances in the DOPRI5 adaptive solver and the range of nodes quantities in the Chebyshev grid. The result of this experiment is presented in Figure 2.3. This plot shows that the smaller tolerance, the more accurate gradients approximation for all considered number of nodes in the Chebyshev grid. At the same time, the larger number of nodes leads to decreasing the approximation quality.

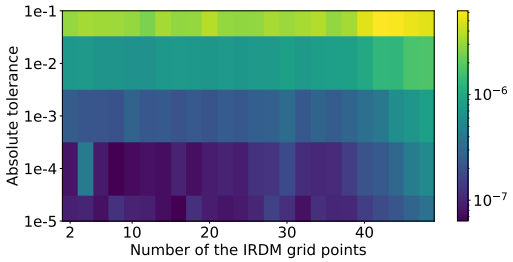


FIGURE 2.3: The dependence of the IRDM gradients error in ℓ_1-norm with respect to the number of nodes in the Chebyshev grid and the tolerance of the DOPRI5 method. The output of the standard backpropagation performed for the DOPRI5 with `1e-7` tolerance was used as a ground truth.

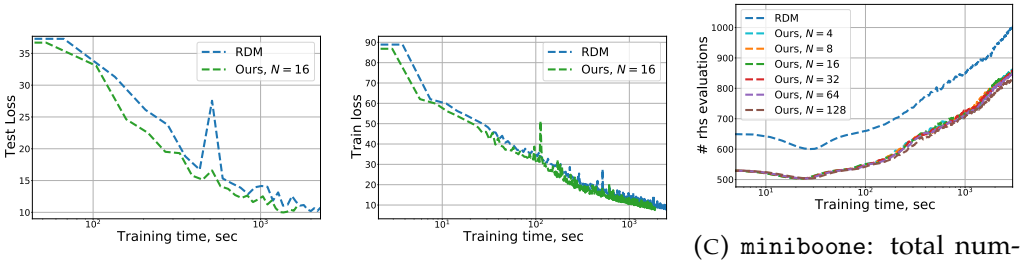
2.5.2 Density Estimation

The problem of density estimation is to reconstruct the probability density function using a set of given data points. We compared the proposed IRDM with the RDM (FFJORD[3] [61] baseline) that exploits the reverse dynamic method to density estimation problem. We tested these methods on four toy datasets (`2spirals`, `pinwheel`, `moons` and `circles`) and tabular `miniboone` dataset [140]. This tabular dataset was used in our experiments since it is large enough and allows considered methods to converge for a reasonable time. According to [61] setting, we stopped the training process if, for the sequential 30 epochs, the test loss does not decrease. Therefore, we excluded test loss values given by the last 30 epochs from the plots. The model for `miniboone` was slightly different from the model from [61]; it includes 10 ODE blocks instead of 1. We used Adam optimizer [96] in all tests on the density estimation problem. For toy datasets, we used the following hyperparameters: learning rate equals 10^{-3}, the number of epochs was 10000, the batch size was 100, absolute and relative tolerances in the DOPRI5 solver were 10^{-5} and 10^{-5}.

Figure 2.4 shows that the test loss decreases more rapidly with our method than with the RDM. To make figures more clear, we plot convergence only for one value of N for every dataset. This value of N gives the best result among the tested values.

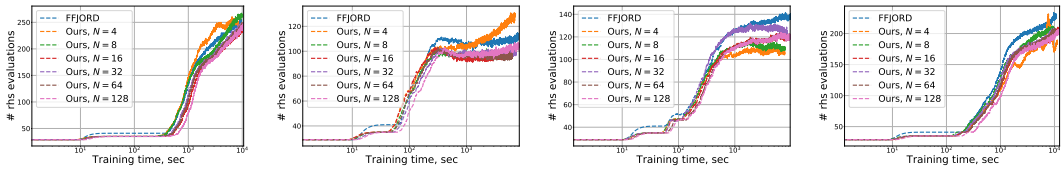
The number of nodes in the Chebyshev grid significantly affects the performance of the proposed method. If this number is small, then the interpolation accuracy is not enough, and the training takes a long time. If this number is large, then the computing of intermediate activations is too costly, and training is relatively slow. In Figure 2.5, the total number of the right-hand side $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ evaluations per training iteration is shown.

[3] https://github.com/rtqichen/ffjord



(A) `miniboone`: test loss (B) `miniboone`: train loss (C) `miniboone`: total number of $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ evaluations

FIGURE 2.4: Comparison of the IRDM with the RDM (baseline from FFJORD) on density estimation problem for tabular dataset `miniboone`. The number of points in the Chebyshev grid N used in the IRDM is given in the legend.

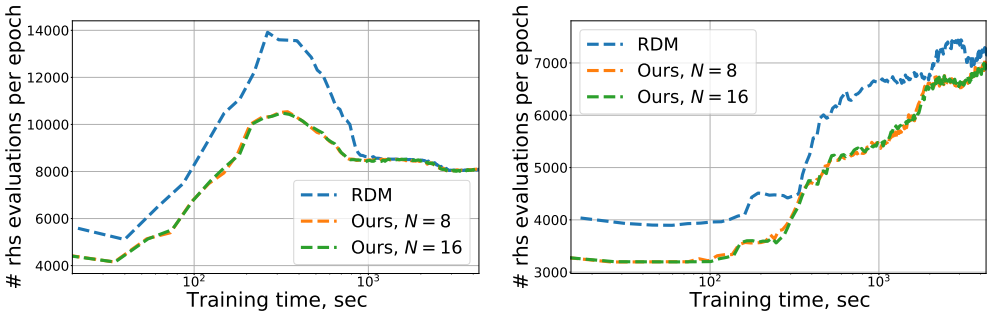


(A) `2spirals` (B) `pinwheel` (C) `moons` (D) `circles`

FIGURE 2.5: Total number of $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ evaluations for density estimation datasets.

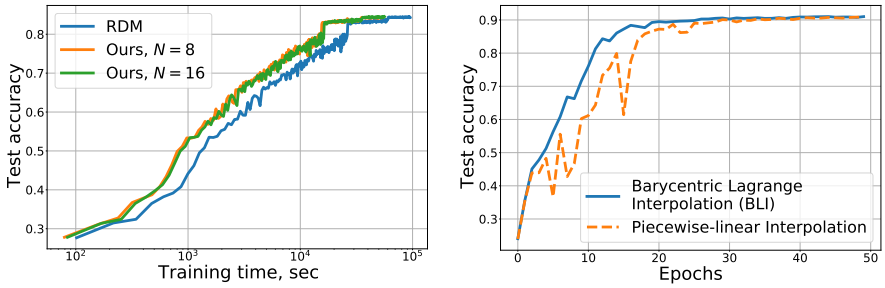
2.5.3 Variational Autoencoder

We also compare the RDM (baselines from FFJORD) and the IRDM on training of variational autoencoder [97]. We use datasets `Caltech` and `Freyfaces`. The test negative ELBO loss and test bits per dim loss are reported for `caltech` and `freyfaces` datasets, respectively. Figure 2.6 illustrates a minor acceleration of convergence provided by the IRDM compared to the RDM. However, the IRDM gives the same final test loss with the same training time as the RDM. We suppose that the reason for such convergence degradation near the optimum is the same as for the density estimation models.



(A) `caltech` (B) `freyfaces`

FIGURE 2.6: Comparison of the number of right-hand side evaluations for the IRDM and the RDM in training variational autoencoder.



(A) Comparison of the IRDM with the RDM in CIFAR10 classification task. IRDM even with $N = 8$ nodes trains faster than RDM.

(B) Comparison of BLI and piecewise-linear interpolation used in the IRDM (8 nodes) in MNIST classification problem.

FIGURE 2.7: Experiments results in the image classification task. The reported values are averaged over three trained models corresponding to the considered tasks.

2.5.4 Classification

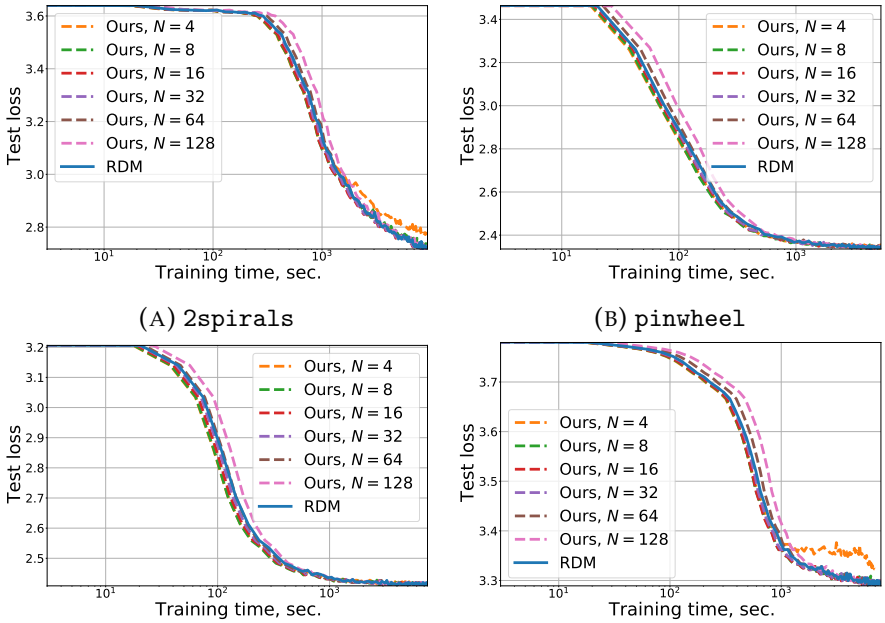
We test the proposed method on the classification problem with the CIFAR10 dataset. The model with a single convolution, a single ODE block, and a linear layer is considered. For this model, the IRDM with 16 points in the Chebyshev grid gives 0.867 test accuracy with the batch size 512 and tolerance `1e-3` in the DOPRI5. We compare the IRDM with the RDM in terms of test loss versus training time. Figure 2.7a demonstrates that the IRDM gives higher test accuracy and requires less training time.

Another experiment is investigating whether other interpolation techniques can be used in the IRDM. We compare the Barycentric Lagrange Interpolation (BLI), which is a default method used in the IRDM, with the piecewise-linear interpolation. We perform 50 epochs in the MNIST classification problem with constant learning rate `1e-1` and without data augmentation. Figure 2.7b confirms our choice of BLI since already after 40 epochs piecewise-linear interpolation provides less stable test accuracy behaviour.

2.5.5 Number of Chebyshev Grid Points

In this section, we study how the quality of a model depends on the number of Chebyshev grid points. To demonstrate this dependence, we perform experiments with a range of N on toy two-dimensional datasets for the density estimation problem. Figures 2.8a and 2.8d show that if the number of nodes is too small, e.g., $N = 4$, the IRDM converges to the higher test loss. It means that the interpolation accuracy is not enough, and a larger number of points in the Chebyshev grid is needed. On the other hand, Figure 2.8 illustrates that if the number of nodes is too large, e.g., $N = 128$, the IRDM might be slower than the RDM. The reason is that a large number of nodes leads to the costly computations of the interpolated activations, see Equation (7) in the main text. Since the right-hand side function in the ODE block for toy datasets is easy to compute, the speedup effect is not much noticeable. However, the total number of the right-hand side function evaluations performed in IRDM

is significantly smaller than in RDM, see Figure 2.9. Therefore, the more computationally expensive the right-hand side function is in the ODE block, the more significant gain one can get from using IRDM.



(A) `2spirals` (B) `pinwheel`

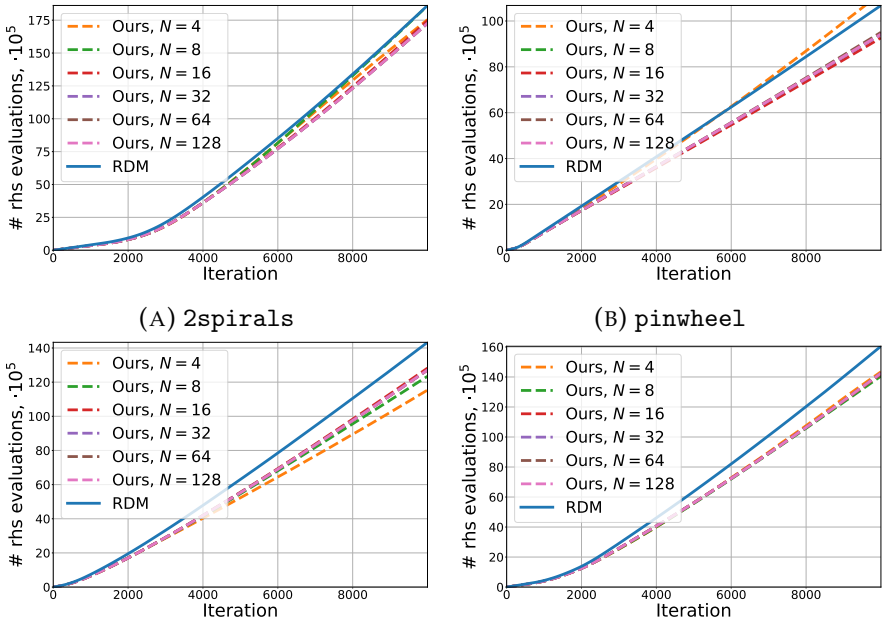
(C) `moons` (D) `circles`

FIGURE 2.8: Comparison of IRDM (our method) and RDM on density estimation problem for toy datasets `2spirals`, `pinwheel`, `moons`, and `circles` in terms of test loss versus wall-clock training time. Comparison results for every dataset are presented in the corresponding subplot. The number of points in Chebyshev grid N used in the IRDM is given in legend.

Table 2.1 shows the total time to perform 10000 training iterations for considered toy datasets. It can be seen that the IRDM with $N = 16$ nodes always outperforms the RDM.

TABLE 2.1: Time (in seconds) to perform 10000 training iterations for toy datasets.

# nodes / dataset	RDM	4	8	16	32	64	128
`pinwheel`	5318	5389	**4642**	4750	4846	5152	5574
`circles`	7874	6927	6864	**6842**	7037	7578	8113
`moons`	7494	**5618**	6062	6471	6518	6720	7192
`2spirals`	9285	8998	9492	**8938**	8947	9267	9680



(A) `2spirals` (B) `pinwheel`

(C) `moons` (D) `circles`

FIGURE 2.9: Comparison of IRDM (our method) and RDM on density estimation problem for toy datasets `2spirals`, `pinwheel`, `moons`, and `circles` in terms of total number of the right-hand side evaluations versus number of iterations. Comparison results for every dataset are presented in the corresponding subplot. The number of points in Chebyshev grid N used in the IRDM is given in legend.

2.6 Conclusion

We have presented the interpolated reverse dynamic method (IRDM) to improve the original reverse dynamic method (RDM) for training neural ODE models. The main idea of IRMD is to reduce the number of ODEs solved during the backward pass by using interpolated values $\boldsymbol{z}(t)$ rather than ones found from equation (2.5). Thus, the total number of right-hand side evaluations during training and convergence time decreases compared to the original reverse dynamic method. We have empirically demonstrated this behavior on density estimation, variational inference, and classification tasks. Also, we have derived a theoretical upper bound on the error in computed gradients induced by the interpolation. The influence of the tolerance in adaptive ODE solver and the number of nodes in the Chebyshev grid is also studied numerically.

Chapter 3

Towards Understanding Normalization in Neural ODEs

3.1 Introduction

Neural Ordinary Differential equations (neural ODEs) are proposed in [22] and model the evolution of hidden representation with ordinary differential equation 2.1. The right-hand side of this ODE is represented with some neural network. If one considers classical Euler scheme to integrate this ODE, then ResNet-like architecture [73] will be obtained. Thus, Neural ODEs are continuous analogue of ResNets. One of the motivation to introduce such models was assumption on smooth evolution of the hidden representation that can be violated with ResNet architecture. Also, in contrast to ResNet models, Neural ODEs share parameters of the ODE right-hand side between steps to integrate this ODE. Thus, Neural ODEs are more memory efficient.

Different normalization techniques were proposed to improve the quality of deep neural networks. Batch normalization [88] is a useful technique when training a deep neural network model. However, it requires computing and storing moving statistics for each time point. It becomes problematic when a number of time steps required for different inputs vary as in recurrent neural networks [81, 29, 7], or the time is continuous as in neural ODEs. We apply different normalization techniques ([148, 117, 7]) to Neural ODE models and report results for the CIFAR-10 classification task. The considered normalization approaches are compared in terms of test accuracy and ability to generalize if a more powerful ODE solver is used in the inference.

3.2 Background

We consider the same IVP as in the previous chapters:

$$\begin{cases} \dfrac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}t} = f(\boldsymbol{z}(t), t, \boldsymbol{\theta}), \quad t \in [t_0, t_1] \\ \boldsymbol{z}(t_0) = \boldsymbol{z}_0, \end{cases} \tag{3.1}$$

where $\boldsymbol{z}_0$ denotes the input features, which are considered as initial value. To solve IVP, we numerically integrate system (3.1) using ODE solver. Depending on the solver type different number of RHS evaluations of (1) are

performed. Initial value problem (3.1) replaces Euler discretization for the same right-hand side that arises in ResNet-like architectures. One part of the standard ResNet-like architecture is the so-called ResNet block, which consists of convolutions, batch normalizations, and ReLU layers. In practice, batch normalization is often used to regularize model, make it more robust, and reduce internal covariate shift [158]. Also, it is shown that batch normalization yields smoother loss surface and makes neural network training faster and more robust [149]. In the context of neural ODEs training, previous studies applied layer normalization [22] and batch normalization [55] but did not investigate the influence of these layers on the model performance. In this study, we focus on the role of normalization techniques in neural ODEs. We assume that proper normalization applied to the layers in ODE blocks leads to the higher test accuracy and smoother dynamic.

According to [112], different problems and neural network architectures require different types of normalization. In our empirical study, we investigate the following normalization techniques to solve the image classification problem with neural ODE models.

- *Batch normalization* (BN; [88]) is the most popular choice for the image classification problem, we discuss its benefits in the above paragraph.
- *Layer normalization* (LN; [7]) and *weight normalization* (WN; [148]) were introduced for RNNs. We consider these normalizations as appropriate candidates for incorporating in neural ODEs since they showed its effectiveness for RNNs that also exploit the idea of weights sharing through time.
- *Spectral normalization* (SN; [117]) was proposed for generative adversarial networks. It is natural to consider SN for neural ODEs since if the Jacobian norm is bounded by 1, one may expect better properties of the gradient propagation in the backward pass.
- We also trained neural ODEs without any normalization (NF).

To perform back-propagation, we use ANODE [55] approach. This is a memory-efficient procedure to compute gradients in neural ODEs with several ODE blocks. This method exploits checkpointing technique at the cost of extra computations.

3.3 Numerical Experiments

This section presents numerical results of applying different normalization techniques to neural ODEs in the CIFAR-10 classification task. Firstly, we compare test accuracy for neural ODE based models with different types of normalizations. Secondly, we present an $(\mathcal{S}, n)$-*criterion* to estimate quality of the trained neural ODE-like model. The source code is available at GitHub repository[1].

[1] https://github.com/juliagusak/neural-ode-norm

In our experiments we consider neural ODE based models, which are built by stacking standard layers and ODE blocks. After replacing ResNet block with ODE block in ResNet4 model, we get

$$\text{conv} \to \textit{norm} \to \text{activation} \to \text{ODE block} \to \text{avgpool} \to \text{fc}$$

an architecture, which we call *ODENet4*. For this model we test different normalization techniques for *norm* layer and inside the ODE block. Similarly, by replacing in ResNet10 architecture ResNet blocks, which do not change spatial size, with ODE blocks, we get the following model:

$$\begin{aligned}\text{conv} \to \textit{norm} \to \text{activation} \to \text{ResNet block} \to \text{ODE block} \to \text{ResNet block} \\ \to \text{ODE block} \to \text{avgpool} \to \text{fc},\end{aligned}$$

which we call *ODENet10*. In contrast to ODENet4, this model admits different normalizations in place of the *norm* layer, inside ResNet blocks and ODE blocks.

We use ANODE to train considered models since it is more robust than the adjoint method (more details see in [55]). In both forward and backward passes through ODE blocks we solve corresponding ODEs using Euler scheme. For the training schedule, we follow the settings from ANODE ([55]). In contrast to ANODEDEV2 ([182]), we include activations and normalization layers to the model. We train considered models for 350 epochs with an initial learning rate equal to 0.1. The learning rate is multiplied by 0.1 at epoch 150 and 300. Data augmentation is implemented. The batch size used for training is 512. For all experiments with different normalization techniques, we use the same settings.

3.3.1 Accuracy

In our experiments, we assume that normalizations for all ResNet blocks are the same, as well as for all ODE blocks. Along with these two normalizations, we vary a normalization technique after the first convolutional layer. We report test accuracy for different normalization schedules for ODENet10. Table 3.1 presents test accuracy given by ODENet10 model. The best model achieves 93% accuracy. It uses batch normalization after the first convolutional layer and in the ResNet blocks, and layer normalization in the ODE blocks. Also, we observe that the elimination of batch normalization after the first convolutional layer and from the ResNet blocks leads to decreasing accuracy to 91.2%. Such quality is even worse than the quality obtained with the model without any normalizations (92%).

3.3.2 $(\mathcal{S}, n)$-criterion of dynamics smoothness in the trained model

Since in neural ODEs like models, we train not only parameters of standard layers, but also parameters on the right-hand side of the system (3.1), the test accuracy is not the only important measure. Another significant criterion is

TABLE 3.1: Comparison of normalization techniques for ODENet10 architecture on CIFAR-10. BN – batch normalization, LN – layer normalization, WN – weight normalization, SN – spectral normalization, NF – the absence of any normalization. To perform back-propagation, we exploit ANODE with a non-adaptive ODE solver. Namely, we use Euler scheme with $N_t = 8$, where N_t is a number of time steps used to solve IVP (3.1). The first row corresponds to the normalization in the ODE blocks. We use BN after the first convolutional layer and inside ResNet blocks, respectively. Standard ResNet10 architecture (only ResNet blocks are used) gives **0.931** test accuracy.

ODE blocks	BN	WN	SN	NF	LN
Accuracy@1	0.762	0.925	0.926	0.927	**0.930**

the smoothness of the hidden representation dynamic that is controlled by the trained parameters of the right-hand side (3.1).

To implicitly estimate this smoothness, we propose an *$(\mathcal{S}, n)$-criterion* that indicates whether more powerful solver induces performance improvement of the trained neural ODE model during evaluation. Here, $\mathcal{S}$ denotes a solver name (Euler, RK2, RK4, etc) and n denotes a number of the right-hand side evaluations necessary for integration of system (3.1), which corresponds to the forward pass through the ODE block. By more powerful solver we mean ODE solver that requires more right-hand side evaluations to solve (3.1) than ODE solver used in training for the same purpose. For example, assume one trains the model with Euler scheme and $n = 2$. Then, we say that ODE block in trained model corresponds to smooth denamics if using Euler scheme with $n > 2$ during evaluation yields higher accuracy. Otherwise, we say that $(\mathcal{S}, n)$-criterion shows the absence of learned smooth dynamics. Worth noting that the $(\mathcal{S}, n)$-criterion has limitation. Namely, it requires the solution of IVP (3.1) to be a Lipschitz function of the right-hand side (3.1) parameters and inputs ([24]). Otherwise, we can not rely on this criterion since the closeness in the right-hand side parameters does not induce the closeness of features that are inputs to the next layers of the model.

In our experiments we consider ODENet4 architecture with four different settings of the Euler scheme: $n = 2, 8, 16, 32$. For each setting we have trained 10 types of architectures that differ from each other by the type of normalization we apply to the first convolutionl layer and convolutional layers in the ODE block. For example, the model named "ODENet4 BN-LN (Euler, 2)" means the following: we have used ODENet4 architecture, where after conv layer follows a BN layer, after each convolutional layer in the right-hand side (3.1) follows an LN layer, and Euler scheme with 2 steps is used to propagate through the ODE block.

For a fixed model trained with (Euler, n_0) solver we check the fulfillment of $(\mathcal{S}, n)$-criterion by evaluating its accuracy with more powerful solver. In this case, we consider the following more powerful solvers: (Euler, n), (RK2, n) and (RK4, n), where $n > n_0$.

In Figure 3.1, we show how test accuracy given by ODENet4 model

with different normalizations changes with varying ODE solvers to integrate IVP (3.1) in ODE blocks. Different line types correspond to different solver type (Euler, RK2, RK4), x-axis depicts the number of the right-hand side evaluations, while y-axis stands for the test accuracy. These models were trained with Euler scheme and after that we use Euler, RK2 and RK4 schemes to compute test accuracy. Every row from top to bottom corresponds to $n = 2, 16, 32$ used in Euler scheme.

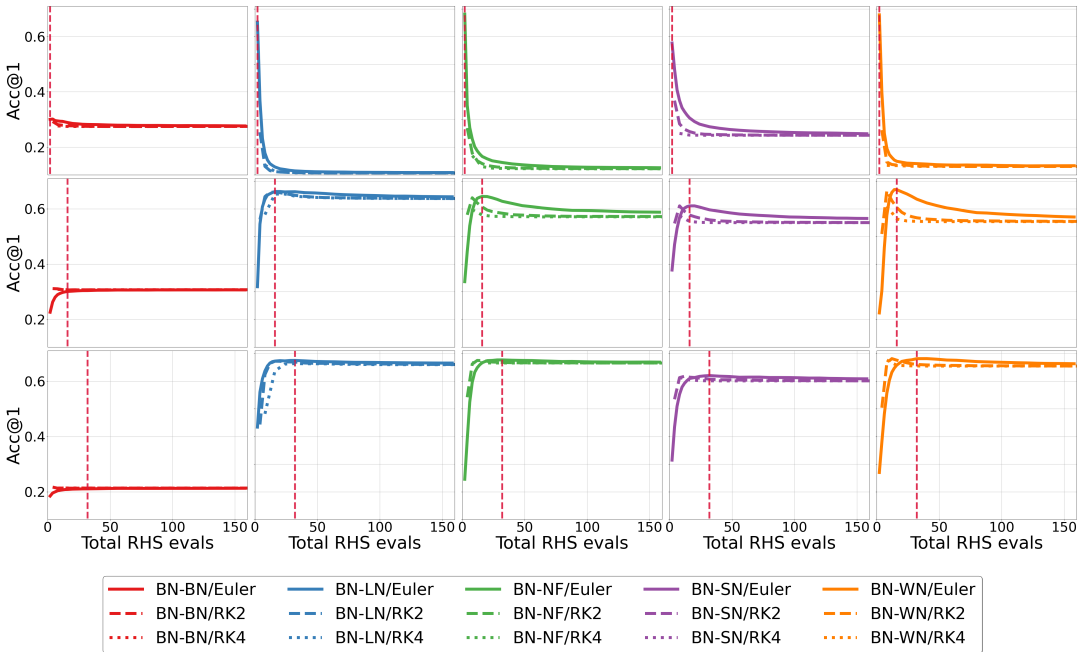


FIGURE 3.1: Illustration of how the choice of ODE solver and normalizations during training implicitly affects the smoothness of learned dynamics. Each subplot corresponds to the model trained with a fixed ODE solver and normalization scheme. Models within one row have the same type of training solver ((Euler, n), $n = 2, 16, 32$ from top to bottom). Models within one column have the same normalization technique. For example, subplot in the third row and the second column corresponds to the ODENet4 model trained with (Euler, 32) solver with BN after the first convolutional layer and LN after convolutional layers inside ODE block. Lines of different style corresponds to different types of test solvers. If model accuracy does not drop when the more powerful ODE solver is used, we conclude that, according to $(\mathcal{S}, n)$-criterion, the model provides a smooth dynamics. For example, the model (Euler, 32) BN-LN trains a smooth dynamics, while (Euler, 2) BN-LN fails to do that. Also, we can observe that to learn the smooth dynamics during training, for some normalization schemes less powerful solvers are required. If we compare BN-LN and BN-WN models, we can see that the first one learns smooth dynamics when Euler with $n = 16$ is used, but the latter one does that only for $n = 32$.

3.4 Discussion and Further research

We have empirically investigated the effect of normalization techniques to ODE based models. For different models, we have compared test accuracy as well as the ability to learn parameters that yield smooth dynamics of hidden representations. Informally, we recommend using layer normalizations or weight normalizations and to avoid batch normalizations for neural ODEs. We have observed that both normalization and type of training solver affect the performance of the final model. Worth noting that pre-trained models, which are close in terms of test accuracy, can significantly differ when it comes to the smoothness of the hidden representations dynamics. Also, we will work on a more rigorous theoretical criterion that can be used to compare ODE based models, considering both neural networks and ODEs metrics.

Chapter 4

Exploring Robustness of Different Solvers for Neural ODEs

4.1 Introduction

In this chapter, we study the neural ODEs from the viewpoint of adversarial robustness. The robustness of neural ODE models to adversarial attacks [3] has already been considered by Hanshu et al. [71]. Authors demonstrate that in the image classification task neural ODE models are more robust to white-box adversarial attacks [57] than CNNs with the similar number of parameters. However, the dependence of the robustness on the used ODE solver in the forward and backward passes is not discussed in [71]. We fill this gap in the presented study and consider the influence of the ODE solver on neural ODE models robustness and test accuracy.

To the best of our knowledge, there were no papers exploring how the choice of a numerical integration scheme affects the quality of the trained neural ODE model. Therefore, we investigate how the choice of ODE solver in the training stage (standard or adversarial) affects the robustness of the trained neural ODE model with respect to the adversarial attacks. We also analyse the possibility to improve the robustness of neural ODE model to black-box attacks through accurate choice of the particular ODE solver in the training stage or use a combination of some set of ODE solvers. Such combination can be obtained according to one of the methods proposed below, see Section 4.2. In addition, we consider how the ODE solver affects the transferability [173] of the adversarial examples generated by neural ODE model.

In this chapter, we consider Runge–Kutta solvers (eq:butcher). The important property of any Runge-Kutta method is the *order of function approximation* denoted by p.

Definition 1 *A Runge-Kutta method is of the order p if the following inequality holds for any $\tilde{t} \in [t_0, t_1]$ such that $\tilde{t} + h \in [t_0, t_1]$: $\|\boldsymbol{z}(\tilde{t} + h) - \hat{\boldsymbol{z}}(\tilde{t} + h)\| \leq Ch^{p+1}$, where $\boldsymbol{z}(t)$ and $\hat{\boldsymbol{z}}(t)$ are ground-truth and approximate dynamics.*

Runge-Kutta methods such that the number of stages s equals to the order p are of particular interest since the corresponding Butcher tableaux can be parametrized with no more than two scalar parameters [169]. We exploit this property when constructing sampling and ensembling solver techniques that operate with a set of solvers rather than with a single predefined one. These

techniques advance the conventional approach to neural ODE training and evaluation by going beyond the use of a predefined solver. To the best of our knowledge we are the first who consider an influence of ODE solvers to adversarial training and robustness if neural ODE models to black-box attacks.

We test the introduced techniques on the image classification tasks (CIFAR10 and MNIST datasets) and robustness to adversarial attacks. Robustness of the trained models to black-box attacks is tested with FGSM [59], PGD [113] and DeepFool [119] attacks from `FoolBox` package [138, 137]. Models to generate black-box attack are taken from `RobustBench` collection of benchmarks [32]. The source code to reproduce results of this study is presented on github[1].

Our main results and contributions to the understanding of neural ODEs properties are summarised as follows:

- We empirically demonstrate that the choice of ODE solver significantly affects robustness of neural ODE models to adversarial attacks. Both standard and adversarial training are tested.
- We propose methods of ODE solvers *sampling* in the training of neural ODEs that lead to more robust trained models without any additional costs.
- We propose methods of ODE solvers *ensembling* in the training of neural ODEs that make the trained model more robust to large attacks.
- We consider approaches to construct models ensembling based on the single trained neural ODE model and multiple ODE solvers.

4.1.1 Related works

Stable neural ODEs training and obtaining competitive results compared with other architectures are challenging tasks that are addressed in many studies. In particular, [55, 185] address the instability of the adjoint method with a checkpointing strategy, and [36] proposes to use the interpolation technique in the backward pass. The importance of the augmentation technique in the context of training neural ODE is presented in [41]. The well-known fact from numerical analysis [169] is that to solve ODE with sufficient accuracy, a small step size in an ODE solver is required. However, this setting leads to an increase of the running time to perform the forward pass in neural ODE. This issue is addressed in [92, 56], where the trained dynamic is forced to be easy to solve with regularization of the loss function and sampling of the integration final time, respectively. Also, the extension of neural ODEs to stochastic neural ODEs is considered in [103, 106, 167, 125].

Besides the standard machine learning quality measures, neural ODEs can be evaluated based on the properties of the learned dynamic. The verification of the learned dynamic stability with respect to decreasing step size in the used ODE solver is studied in [66] where $(\mathcal{S}, n)$-criterion is proposed for

[1] `https://github.com/SamplingAndEnsemblingSolvers/SamplingAndEnsemblingSolvers`

that purpose. Further, similar analysis of the learned dynamic is performed in [126]. The related question on the importance of control trained dynamic properties is discussed in [135], where the continuous-in-depth extension of ResNet architecture is proposed.

One of the crucial factors in the evaluation of machine learning models is robustness to adversarial attacks [3]. Adversarial example is an input image that is modified by adding perturbation, which is almost invisible for the human eye, so called adversarial perturbation. We say that an adversarial attack takes place if an adversarial example fools predictive model, i.e. model gives wrong class label. Adversarial examples can be generated and tested on different models. Therefore, concepts of white-box and black-box attacks appear. The adversarial attack is called *white-box*, if all knowledge about the model (parameters, loss, architecture, data labels) is used to generate perturbation. In contrast, the attack is called *black-box* if none of the knowledge about model is used. The intermediate case such that the limited knowledge about model may be involved, except its parameters, is called *semi-black-box* attack. The model to generate an adversarial example is called a source model, and the model to evaluate the label for adversarial example is called a target model. If the gradient of loss function with respect to the input is used to calculate perturbation, we say that the attack is gradient-based [128]. The classical examples of such attacks are FGSM [59] and PGD [113]. Other approaches to compute adversarial examples are DeepFool [119] and Carlini-Wagner attack [17]. The studies [71, 19] demonstrate that neural ODEs are more robust to adversarial attacks than classical CNN models.

Key idea: Runge-Kutta methods typically used in the neural ODE training can be parametrized with no more than two scalar variables.

In this chapter, we consider explicit Runge-Kutta methods such that their order p equals to the number of stages s. This requirement leads to the constraints on the coefficients from Butcher tableau. These constraints induce parametrizations of the Runge-Kutta methods that we will use to make neural ODE model more robust. We provide the considered parametrizations of Runge-Kutta methods below following [169].

Runge-Kutta methods of the 2-nd order with two stages. These Runge-Kutta methods are defined by the Butcher tableau whose coefficients have to satisfy the following system of equations:

$$\begin{cases} b_1 + b_2 = 1 \\ b_2 c_2 = \frac{1}{2}. \end{cases}$$

Thus, these methods can be parametrized by a single parameter $u \in (0, 1]$; see the corresponding Butcher table (4.1a).

Definition 2 *Let parameters of Runge-Kutta methods be a set of values that uniquely define the Butcher tableau corresponding to the considered class of Runge-Kutta methods.*

In particular, midpoint rule and Heun's method are particular cases of such parametrization if $u = \frac{1}{2}$ and $u = 1$, respectively, see Tables 4.1b and 4.1c.

TABLE 4.1: 2-stage RK method of the 2-nd order

0	0	
u	u	0
	$1 - \frac{1}{2u}$	$\frac{1}{2u}$

(A) Parametrized Butcher tableau

0	0	
$\frac{1}{2}$	$\frac{1}{2}$	0
	0	1

(B) Midpoint rule

0	0	
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

(C) Heun's method

Thus, tuning of parameter u leads to different solvers of the same order that can be combined in training of neural ODE models. Techniques to combine such Runge-Kutta methods are presented in Section 4.2.

4.2 Meta Neural ODE

Key idea. Solver parameters are modified during the neural ODE training by sampling from a given distribution. Hence, the model is trained using a large set of Runge-Kutta solvers instead of a single one, and yields better robustness to adversarial attacks without time overhead.

Since we want to adjust an ODE solver during training to improve neural ODE, the natural idea is to compute gradient of the loss function with respect to solver parameters and update them according to the gradient method altogether with weights in other layers. We have tested this approach and figured out that the training is quite unstable since the feasible parameters are not arbitrary and their clamping does not lead to desired improvement. Thus, in this chapter we introduce gradient-free methods of updating Runge-Kutta solver parameters during training.

Solver sampling. Solver sampling technique can be splitted in two methods: switching and smoothing. During the neural ODE training, at each epoch we randomly choose a solver from a pre-defined set of solvers to perform propagation through the model. If the set of solvers is continuous, we call this strategy *solver smoothing*, otherwise, we refer to it as *solver switching*. The latter approach requires a pre-defined set of parameterizations, each of which corresponds to one solver. Considering s-stage Runge-Kutta methods of order $p \leq 4$, $p = s$, each parametrization corresponds to one or two scalar values.

The sampling of solver parameters can be done uniformly or according to some prior fixed distribution. The benefits of switching is that it does not lead to computational overhead comparing to the single solver while trying to make the model more robust to the choice of the solver. However, such regime might make a neural ODE training via backpropogation difficult, if we have a limited number of solvers that exhibit different dynamics. That leads as to the smoothing approach, which can be considered as a continuous case of switching. Smoothing regime requires to set in advance a parameterization of one initial solver. During the training, parameters for the next solver

are sampled from a continuous distribution, whose mean corresponds with parameters of the initial solver. We expect that this approach leads to *smoothing* of the trained dynamics and make it more robust to adversarial attacks.

Ensembling of solvers outputs. Another approach to adjust ODE solver during training neural ODE models is *ensembling of solvers outputs*. The idea of this method is to set ODE solvers corresponding to the same parameterization but from different values of parameters, compute trajectories with these solvers and average the computed trajectories with some pre-defined weights. The resulting dynamic approximation takes into account dynamics generated by different ODE solvers and therefore is more robust to the ODE solver choice. However, this method requires extra costs since multiple trajectories are computed. That is why we consider this approach as additional.

Ensemble of models for free. When training a neural ODE using smoothing regime, we end up with a model, which performs well on a given task for a family of solvers. Hence, we can use this fact to build an ensemble of models to further improve the performance.

4.3 Experiments

4.3.1 Motivation to explore solver parameterizations

In this section, we provide a motivation experiment to explore the influence of solver parameterizations on neural ODEs performance. In [71] the authors pointed out that neural ODE models are more robust than CNN models for image classification tasks. Inspired by this chapter, we move further and consider the influence of different ODE solvers on the accuracy of models when the samples are modified by an adversarial attack. We call this accuracy a *robust accuracy*.

We use the classification task on the MNIST dataset to illustrate the dependency of the robust accuracy on the choice of ODE solver. We consider the 2-nd order two stages Runge-Kutta methods for various values of parameter u from the interval $(0, 1]$. The corresponding Butcher tableau is given in Table 4.1a. The model we train is sequentially stacked three blocks: block of standard neural network layers, ODE block, and another block of neural network layers.

We evaluate the robust accuracy using PGD attack with $\varepsilon = 0.3$, learning rate $2/255$ and 7 iterations. We provide the dependency of the robust accuracy on the value of solver parameter u in Figure 4.1. We see from Figure 4.1 that models trained with different solver parameterizations yield different robust accuracies while maintaining similar standard accuracy.

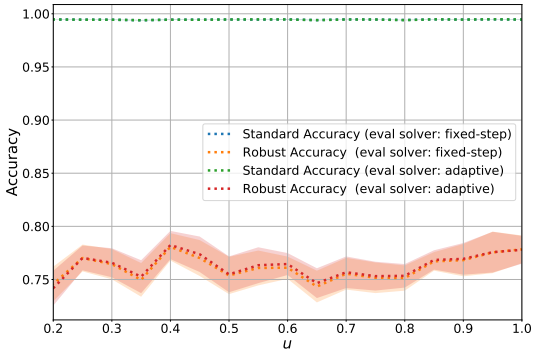


FIGURE 4.1: Robust accuracy of the model on MNIST dataset vs. different values of parameter u in the 2-nd order Runge-Kutta solver (see Table 4.1).

4.3.2 Adversarial training on CIFAR-10

Motivated by the experiment from Section 4.3.1 we consider adversarial training of Neura ODE with Meta ODE blocks on the CIFAR-10 image classification task. We experiment with an architecture of the following type: (Conv layer $\rightarrow$ PreResNet block $\rightarrow$ Meta ODE block $\rightarrow$ PreResNet block $\rightarrow$ Meta ODE block $\rightarrow$ GeLU $\rightarrow$ AveragePooling $\rightarrow$ FullyConnected Layer). We trained the model using the technique proposed in [171]. As optimizer, we use SGD with a momentum 0.9 and a Cyclic LR schedule (one cycle per 36 epochs with batch size 256). We use fixed-step size ODE solvers to propagate through the Meta ODE block.

In the following Sections 4.3.4, 4.3.3 we provide results for the resistance to adversarial attacks of Neural ODEs trained with different Meta ODE block regimes. We consider black-box attack scenarios. To generate test attacks we use Foolbox [138, 137] package. We use pre-trained models from RobustBench to generate adversarial examples for black-box experiments.

Training with a pre-defined (standalone) solver. We have trained the described architecture several times using different 8-step standalone solvers, namely, Euler solver and 2-stage 2-nd order Runge-Kutta solver with $u = 0.5$.

Training with solver switching. Using 2-stage Runge-Kutta solvers of 2-nd and 4-th order, we built several groups of solvers (each group contained 2, 3, 10, or 20 solvers of the same order). We trained the same neural ODE architecture using Meta ODE block in switching regime: at each iteration, one solver from the group was chosen to perform propagation.

Training with solver smoothing. We chose 2-stage 2-nd order Runge-Kutta solvers for our solver smoothing strategy. At each iteration, we use one solver to propagate through the Meta ODE block. The parameter u of the solver is sampled from the normal distribution $\mathcal{N}(0.5, 0.0125)$.

Training with solver ensembling. We consider a group of 2-nd order Runge-Kutta solvers.

4.3.3 Neural Networks attack Neural ODEs

In black-box experiments, we use convolutional neural networks to generate adversarial attacks and measure robust accuracy of neural ODEs trained with Meta ODE block in standalone, sampling (both switching and smoothing), and ensembling regimes. To validate all pre-trained architectures we use 2-stage 2-nd order Runge-Kutta solver in standalone regime with $u = 0.5$ and 8 steps (Tables 4.2, 4.4).

Neural ODEs trained in standalone regime. Table 4.4 shows that the usage of 2-nd order Runge-Kutta solver during training yields models with better resistance to black-box attacks than the usage of Euler solver. Our proposed solver regimes further improve these results.

Neural ODEs trained in switching regime. We found that the resulting robust accuracy is approximately the same for models trained with groups of solvers of the same order, and it is higher for higher-order Runge-Kutta solvers. Table 4.2 compares the influence of different solver regimes (all regimes use 2-nd order 2-stage Runge-Kutta solvers) to the robustness of the trained model. We see that solver switching outperforms other regimes when ε is small.

Neural ODEs trained in smoothing/ensembling regime. From Table 4.2 we can see that the solver ensembling regime can be helpful for high values of perturbations, and the smoothing regime yields the best robustness in most cases.

4.3.4 Neural ODEs attack Neural ODEs

In this section, we consider the case, when we know everything about the attacked neural ODE except the solver used during training. We refer to this case as *grey-box attacks*. Table 4.3 compares models pre-trained using standalone, switching, smoothing and ensembling regimes (with the same setting as in previous sections). We can see the from Table 4.3 that smoothing and ensembling regimes lead to more robust models w.r.t FGSM and PGD attacks. And that DeepFool attack is the most sensitive one to the difference in the solvers used to perform propagation through the source and target neural ODEs.

4.4 Conclusion

In this chapter, we show that in the adversarial tasks, the performance of neural ODEs depends on the choice of an ODE solver. We consider different parameterizations of the standard Runge-Kutta methods and study its influence on the neural ODE models training. We propose techniques of sampling (switching/smoothing) and ensembling of ODE solvers during propagation through neural ODEs that lead to more robust models. We validated models trained with our methods using FGSM, PGD and DeepFool attacks. We observed that for different size of attacks and different neural network architectures our proposed techniques improves the resistance of the resulting model to adversarial attacks. The presented approach can be extended to other

that Runge-Kutta parameterizations of ODE solvers and may be investigated in different applications.

TABLE 4.2: Blackbox attacks on CIFAR10 Neural ODE models. Source models are from RobustBench [32] and papers by Carmon et al. [18], Sehwag et al. [152], and Wong, Rice, and Kolter [171]. PGD attack is performed with 20 iterations and DeepFool attack is performed with 50 iterations.

Source model	255ϵ	Attack	Ensembling	Smoothing	Standalone	Switching
Carmon et al. [18]	2	DeepFool	79.77 ± 0.47	$\mathbf{80.12 \pm 0.14}$	79.93 ± 0.29	80.06 ± 0.21
Carmon et al. [18]	4	DeepFool	77.05 ± 0.27	$\mathbf{77.43 \pm 0.22}$	77.14 ± 0.2	77.31 ± 0.15
Carmon et al. [18]	8	DeepFool	71.47 ± 0.3	$\mathbf{71.62 \pm 0.37}$	71.31 ± 0.21	71.21 ± 0.22
Carmon et al. [18]	16	DeepFool	60.87 ± 0.16	$\mathbf{60.95 \pm 0.44}$	60.8 ± 0.6	60.68 ± 0.2
Carmon et al. [18]	32	DeepFool	$\mathbf{48.44 \pm 0.47}$	48.39 ± 0.47	48.23 ± 0.55	48.18 ± 0.26
Carmon et al. [18]	2	FGSM	79.0 ± 0.39	79.28 ± 0.1	79.07 ± 0.35	$\mathbf{79.42 \pm 0.2}$
Carmon et al. [18]	4	FGSM	75.43 ± 0.39	75.75 ± 0.06	75.29 ± 0.14	$\mathbf{75.75 \pm 0.1}$
Carmon et al. [18]	8	FGSM	67.64 ± 0.41	$\mathbf{67.85 \pm 0.23}$	67.24 ± 0.26	67.38 ± 0.29
Carmon et al. [18]	16	FGSM	53.1 ± 0.34	$\mathbf{53.18 \pm 0.26}$	52.85 ± 0.24	52.75 ± 0.2
Carmon et al. [18]	32	FGSM	35.54 ± 0.12	$\mathbf{35.73 \pm 0.04}$	35.15 ± 0.57	35.26 ± 0.48
Carmon et al. [18]	2	PGD	78.82 ± 0.45	79.11 ± 0.1	78.87 ± 0.34	$\mathbf{79.24 \pm 0.17}$
Carmon et al. [18]	4	PGD	74.51 ± 0.38	74.85 ± 0.19	74.45 ± 0.29	$\mathbf{74.98 \pm 0.27}$
Carmon et al. [18]	8	PGD	63.83 ± 0.35	$\mathbf{64.0 \pm 0.36}$	63.53 ± 0.17	63.65 ± 0.31
Carmon et al. [18]	16	PGD	$\mathbf{48.21 \pm 0.13}$	48.16 ± 0.08	47.82 ± 0.41	47.78 ± 0.28
Carmon et al. [18]	32	PGD	$\mathbf{35.36 \pm 0.1}$	35.24 ± 0.21	34.88 ± 0.27	34.97 ± 0.63
Sehwag et al. [152]	2	DeepFool	79.77 ± 0.36	$\mathbf{80.28 \pm 0.07}$	79.87 ± 0.34	80.15 ± 0.18
Sehwag et al. [152]	4	DeepFool	77.0 ± 0.31	$\mathbf{77.47 \pm 0.1}$	76.98 ± 0.37	77.41 ± 0.06
Sehwag et al. [152]	8	DeepFool	70.95 ± 0.35	$\mathbf{71.65 \pm 0.19}$	70.99 ± 0.26	71.36 ± 0.14
Sehwag et al. [152]	16	DeepFool	60.63 ± 0.19	$\mathbf{61.12 \pm 0.5}$	60.2 ± 0.35	60.89 ± 0.07
Sehwag et al. [152]	32	DeepFool	50.75 ± 0.11	$\mathbf{51.27 \pm 0.49}$	50.26 ± 0.53	50.86 ± 0.25
Sehwag et al. [152]	2	FGSM	78.06 ± 0.36	78.37 ± 0.06	78.17 ± 0.4	$\mathbf{78.46 \pm 0.13}$
Sehwag et al. [152]	4	FGSM	73.44 ± 0.28	$\mathbf{73.66 \pm 0.14}$	73.09 ± 0.33	73.55 ± 0.29
Sehwag et al. [152]	8	FGSM	62.53 ± 0.27	$\mathbf{62.84 \pm 0.14}$	62.27 ± 0.29	62.46 ± 0.39
Sehwag et al. [152]	16	FGSM	$\mathbf{43.4 \pm 0.33}$	43.33 ± 0.25	42.93 ± 0.31	43.11 ± 0.08
Sehwag et al. [152]	32	FGSM	24.64 ± 0.33	$\mathbf{24.81 \pm 0.51}$	24.04 ± 0.43	24.45 ± 0.27
Sehwag et al. [152]	2	PGD	78.0 ± 0.29	78.3 ± 0.09	78.07 ± 0.44	$\mathbf{78.39 \pm 0.19}$
Sehwag et al. [152]	4	PGD	72.96 ± 0.28	$\mathbf{73.18 \pm 0.16}$	72.67 ± 0.34	73.11 ± 0.14
Sehwag et al. [152]	8	PGD	59.86 ± 0.11	$\mathbf{59.88 \pm 0.22}$	59.76 ± 0.26	59.65 ± 0.19
Sehwag et al. [152]	16	PGD	$\mathbf{40.66 \pm 0.3}$	40.64 ± 0.19	40.51 ± 0.24	40.59 ± 0.16
Sehwag et al. [152]	32	PGD	$\mathbf{27.78 \pm 0.5}$	27.76 ± 0.27	27.4 ± 0.44	27.62 ± 0.36
Wong, Rice, and Kolter [171]	2	DeepFool	79.62 ± 0.31	$\mathbf{80.14 \pm 0.17}$	79.63 ± 0.37	80.07 ± 0.14
Wong, Rice, and Kolter [171]	4	DeepFool	76.74 ± 0.28	$\mathbf{77.31 \pm 0.48}$	76.71 ± 0.32	77.13 ± 0.24
Wong, Rice, and Kolter [171]	8	DeepFool	71.53 ± 0.38	$\mathbf{71.9 \pm 0.65}$	71.3 ± 0.19	71.66 ± 0.3
Wong, Rice, and Kolter [171]	16	DeepFool	63.96 ± 0.56	$\mathbf{64.3 \pm 0.95}$	63.16 ± 0.4	64.07 ± 0.32
Wong, Rice, and Kolter [171]	32	DeepFool	58.92 ± 0.81	$\mathbf{59.2 \pm 1.04}$	57.95 ± 0.28	58.94 ± 0.45
Wong, Rice, and Kolter [171]	2	FGSM	77.35 ± 0.3	77.65 ± 0.04	77.25 ± 0.34	$\mathbf{77.77 \pm 0.01}$
Wong, Rice, and Kolter [171]	4	FGSM	71.66 ± 0.31	$\mathbf{71.9 \pm 0.27}$	71.28 ± 0.19	71.69 ± 0.2
Wong, Rice, and Kolter [171]	8	FGSM	58.63 ± 0.12	$\mathbf{58.79 \pm 0.27}$	58.67 ± 0.47	58.59 ± 0.1
Wong, Rice, and Kolter [171]	16	FGSM	37.33 ± 0.36	$\mathbf{37.42 \pm 0.54}$	36.87 ± 0.37	37.01 ± 0.32
Wong, Rice, and Kolter [171]	32	FGSM	17.95 ± 0.23	$\mathbf{18.05 \pm 0.42}$	17.51 ± 0.42	18.05 ± 0.18
Wong, Rice, and Kolter [171]	2	PGD	77.27 ± 0.32	77.6 ± 0.02	77.2 ± 0.38	$\mathbf{77.69 \pm 0.07}$
Wong, Rice, and Kolter [171]	4	PGD	71.17 ± 0.18	$\mathbf{71.38 \pm 0.24}$	70.91 ± 0.22	71.34 ± 0.21
Wong, Rice, and Kolter [171]	8	PGD	55.63 ± 0.29	$\mathbf{55.97 \pm 0.02}$	55.71 ± 0.4	55.48 ± 0.08
Wong, Rice, and Kolter [171]	16	PGD	34.49 ± 0.24	$\mathbf{34.51 \pm 0.2}$	34.01 ± 0.31	34.15 ± 0.42
Wong, Rice, and Kolter [171]	32	PGD	20.62 ± 0.28	$\mathbf{20.64 \pm 0.34}$	20.44 ± 0.48	20.55 ± 0.44

TABLE 4.3: Greybox attacks for $\varepsilon = 8/255$. Mean robust accuracy and standard errors averaged over three runs are reported below. Parameters for PGD attack are the following: number of steps is 7, step size is 2/255. Maximum number of steps in DeepFool attack is 50.

Model \ Attack	FGSM	PGD	DeepFool
$N = 8$		$N = 8$	
Standalone	40.74 ± 0.1	34.49 ± 0.08	36.34 ± 0.11
Switching	40.89 ± 0.08	34.88 ± 0.04	36.37 ± 0.12
Smoothing	41.39 ± 0.07	35.17 ± 0.1	36.85 ± 0.07
Ensembling	41.25 ± 0.13	35.13 ± 0.12	36.88 ± 0.11
$N = 8$		$N = 32$	
Standalone	40.72 ± 0.11	34.44 ± 0.1	47.02 ± 0.04
Switching	40.89 ± 0.07	34.85 ± 0.04	47.62 ± 0.17
Smoothing	41.37 ± 0.05	35.22 ± 0.09	47.22 ± 0.27
Ensembling	41.22 ± 0.14	35.16 ± 0.13	47.45 ± 0.12

(A) Solver to generate attack and to compute robust accuracy is RK2, $u = 0.5$.

Model \ Attack	FGSM	PGD	DeepFool
$N = 8$		$N = 8$	
Standalone	40.73 ± 0.1	34.49 ± 0.09	36.37 ± 0.13
Switching	40.93 ± 0.08	34.89 ± 0.05	36.4 ± 0.13
Smoothing	41.36 ± 0.06	35.19 ± 0.11	36.84 ± 0.05
Ensembling	41.25 ± 0.11	35.15 ± 0.14	36.85 ± 0.1
$N = 8$		$N = 32$	
Standalone	40.69 ± 0.1	34.42 ± 0.1	47.87 ± 0.17
Switching	40.88 ± 0.07	34.88 ± 0.04	48.44 ± 0.12
Smoothing	41.37 ± 0.06	35.21 ± 0.09	47.95 ± 0.31
Ensembling	41.19 ± 0.14	35.14 ± 0.12	47.96 ± 0.2

(B) Solver to generate attack and to compute robust accuracy is RK2, $u = 1$.

Model \ Attack	FGSM	PGD	DeepFool
$N = 8$		$N = 8$	
Standalone	40.75 ± 0.1	34.47 ± 0.09	41.74 ± 0.25
Switching	40.91 ± 0.08	34.89 ± 0.04	41.4 ± 09.03
Smoothing	41.37 ± 0.08	35.19 ± 0.1	41.4 ± 0.01
Ensembling	41.23 ± 0.13	35.15 ± 0.12	41.63 ± 0.31
$N = 8$		$N = 32$	
Standalone	40.69 ± 0.1	34.43 ± 0.11	47.96 ± 0.19
Switching	40.89 ± 0.07	34.87 ± 0.04	48.57 ± 0.1
Smoothing	41.36 ± 0.06	35.21 ± 0.09	48.02 ± 0.28
Ensembling	41.18 ± 0.14	35.15 ± 0.12	48.07 ± 0.23
$N = 32$		$N = 32$	
Standalone	40.73 ± 0.11	34.42 ± 0.09	36.43 ± 0.13
Switching	40.96 ± 0.06	34.88 ± 0.04	36.52 ± 0.11
Smoothing	41.39 ± 0.08	35.21 ± 0.09	36.98 ± 0.05
Ensembling	41.29 ± 0.14	35.18 ± 0.12	36.89 ± 0.13

(C) Solver to generate attack is RK2, $u = 1$. Solver to compute robust accuracy is RK2, $u = 0.5$.

Model \ Attack	FGSM	PGD	DeepFool
$N = 8$		$N = 8$	
Standalone	40.76 ± 0.1	34.5 ± 0.09	40.95 ± 0.05
Switching	40.91 ± 0.06	34.87 ± 0.04	40.65 ± 0.2
Smoothing	41.38 ± 0.05	35.18 ± 0.11	41.03 ± 0.19
Ensembling	41.25 ± 0.12	35.15 ± 0.13	40.99 ± 0.07
$N = 8$		$N = 32$	
Standalone	40.72 ± 0.11	34.44 ± 0.1	47.0 ± 0.04
Switching	40.89 ± 0.06	34.86 ± 0.04	47.53 ± 0.17
Smoothing	41.37 ± 0.06	35.22 ± 0.09	47.17 ± 0.25
Ensembling	41.21 ± 0.14	35.16 ± 0.13	47.42 ± 0.13
$N = 32$		$N = 32$	
Standalone	40.72 ± 0.11	34.42 ± 0.09	36.42 ± 0.14
Switching	40.96 ± 0.06	34.88 ± 0.04	36.48 ± 0.11
Smoothing	41.4 ± 0.08	35.21 ± 0.09	36.97 ± 0.05
Ensembling	41.29 ± 0.14	35.18 ± 0.12	36.93 ± 0.14

(D) Solver to generate attack is RK2, $u = 0.5$. Solver to compute robust accuracy is RK2, $u = 1$.

TABLE 4.4: Comparison of adversarial robustness of the RK2 and Euler solvers.

	FGSM-8/255		PGD-8/255		DeepFool-8/255	
Source Model	Euler	RK2	Euler	RK2	Euler	RK2
Sehwag et al. [152]	62.0 ± 0.12	$\mathbf{62.27 \pm 0.1}$	59.38 ± 0.19	$\mathbf{59.76 \pm 0.09}$	70.78 ± 0.16	$\mathbf{70.99 \pm 0.09}$
Wong, Rice, and Kolter [171]	58.21 ± 0.05	$\mathbf{58.67 \pm 0.16}$	55.27 ± 0.1	$\mathbf{55.71 \pm 0.13}$	71.01 ± 0.17	$\mathbf{71.3 \pm 0.06}$
Carmon et al. [18]	67.06 ± 0.1	$\mathbf{67.24 \pm 0.09}$	63.21 ± 0.12	$\mathbf{63.53 \pm 0.06}$	70.94 ± 0.2	$\mathbf{71.31 \pm 0.07}$
	FGSM-16/255		PGD-16/255		DeepFool-16/255	
Source Model	Euler	RK2	Euler	RK2	Euler	RK2
Sehwag et al. [152]	42.73 ± 0.02	$\mathbf{42.93 \pm 0.1}$	40.08 ± 0.15	$\mathbf{40.51 \pm 0.08}$	60.39 ± 0.13	$\mathbf{60.2 \pm 0.12}$
Wong, Rice, and Kolter [171]	36.66 ± 0.18	$\mathbf{36.87 \pm 0.12}$	33.96 ± 0.1	$\mathbf{34.01 \pm 0.1}$	63.31 ± 0.22	$\mathbf{63.16 \pm 0.13}$
Carmon et al. [18]	52.33 ± 0.12	$\mathbf{52.85 \pm 0.08}$	47.25 ± 0.16	$\mathbf{47.82 \pm 0.14}$	60.24 ± 0.21	$\mathbf{60.8 \pm 0.2}$

Chapter 5

Reduced-Order Modeling of Deep Neural Networks

5.1 Introduction

Recent studies [22, 61, 66, 36] have shown the connection between deep neural networks and systems of ordinary differential equations (ODE). In these works, the output of the layer during the forward pass was treated as the state of a dynamical system at a given time. One of the effective methods for accelerating computations in dynamical systems is the construction of reduced models [134]. The classical approach for building such models is the Discrete Empirical Interpolation Method (DEIM; see [21]). The idea of DEIM is based on a low-dimensional approximation of the state vector, combined with efficient recalculation of the coefficients in this low-dimensional space through the selection of the submatrix of sufficiently large volume.

In this chapter, we use the above connection to build a reduced model of deep neural network for a given pre-trained (fully-connected or convolutional) network. We call this model Reduced-Order Network (RON). The reduced model is a fully-connected network that has smaller computational complexity than the original neural network. The complexity is defined as the number of floating-point operations (FLOP) required to propagate through the network. Thus, the inference of RON can be faster.

Following the reduced-order modeling approach, we assume that the outputs of some layers lie in low-dimensional subspaces. We will refer to this assumption as the *low-rank assumption*. Let $\boldsymbol{x}$ be the object from the dataset, and $\boldsymbol{z}_k = \boldsymbol{z}_k(\boldsymbol{x})$ be the vectorized output of the k-th layer. We assume that there exists a matrix $\boldsymbol{V}_k \in \mathbb{R}^{D_k \times R_k}$ $(D_k \gg R_k)$ such that

$$\boldsymbol{z}_k \cong \boldsymbol{V}_k \boldsymbol{c}_k, \tag{5.1}$$

where $\boldsymbol{c}_k = \boldsymbol{c}_k(\boldsymbol{x})$ are embeddings. The matrix $\boldsymbol{V}_k$ is the same for all $\boldsymbol{x}$.

This simple linear representation itself can not help to reduce the complexity of neural networks, because all linear operations in a neural network are followed by non-linear element-wise functions. However, we propose how to approximate the next embedding based on the previous one.

As a result, under the low-rank assumption most fully-connected and

convolutional neural networks[1] can be approximated by fully-connected networks with a smaller number of processing units. In other words, instead of dealing with huge feature maps, we project the input of the entire network into a low-dimensional space and then operate with low-dimensional representations. We restore the output dimensionality of the model using a linear transformation. As a result, the complexity of neural networks can be significantly decreased.

Even if the low-rank assumption holds only very approximately, we still can use it to initialize a new network and then perform several iterations of fine-tuning.

Our main contributions are:

- We propose a new low-rank training-free method for speeding up the inference of pre-trained deep neural networks and show how to efficiently use the rectangular maximum volume algorithm to reduce the dimensionality of layers and estimate the approximation error.
- We validate and evaluate performance the proposed approach in a series of computational experiments with LeNet pre-trained on MNIST and VGG models pre-trained on CIFAR-10/CIFAR-100/SVHN.
- We show that our method works well on top of pruning techniques and allows us to speed up the models that have already been accelerated.

5.2 Background

In this section, we give a brief description of the rectangular volume algorithm (Subsection 5.2.1) and explain how to compute low-dimensional subspaces of embeddings (Subsection 5.2.2). This information is required to clearly understand what follows.

5.2.1 Maximum Volume Algorithm and Sketching

The rectangular maximum volume algorithm [116] is a greedy algorithm that searches for a maximum volume submatrix of a given matrix. The volume of a matrix $\boldsymbol{A}$ is defined as

$$\text{vol}\,(A) = \det(\boldsymbol{A}^\top \boldsymbol{A}). \tag{5.2}$$

This algorithm has several practical applications [46, 116]. In this chapter, we use it to reduce the dimensionality of overdetermined systems as follows.

Assume, $\boldsymbol{A} \in \mathbb{R}^{D\times R}$ is a tall-and-thin matrix ($D \gg R$); and we have to solve a linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{5.3}$$

[1] We mean convolutional neural networks consisting of convolutions, non-decreasing activation functions, batch normalizations, maximum poolings, and residual connections.

with a fixed matrix $\boldsymbol{A}$ for an arbitrary right-hand side $\boldsymbol{b} \in \mathbb{R}^D$. The solution is typically given by

$$\boldsymbol{x} = \boldsymbol{A}^{\dagger}\boldsymbol{b}, \tag{5.4}$$

where $\boldsymbol{A}^{\dagger} = (\boldsymbol{A}^{\top}\boldsymbol{A})^{-1}\boldsymbol{A}^{\top}$ is the Moore-Penrose pseudoinversion of $\boldsymbol{A}$. The issue is that a matrix-by-vector product with $R \times D$ matrix $\boldsymbol{A}^{\dagger}$ costs too much. Moreover, for the ill-conditioned matrix, the solution is not very stable.

Instead of using all D equations, we can select the most "representative" of them. urpose, we apply the rectangular maximum volume algorithm[2] to the matrix $\boldsymbol{A}$. It returns a set P row indices ($R \leq P \ll D$), which corresponds to equations used for further calculations. In this work, we choose P on the segment $[R, 2R]$.

A submatrix consisting of P given rows can be viewed as $\boldsymbol{SA}$, where $\boldsymbol{S} \in \{0,1\}^{P \times D}$. We call $\boldsymbol{S}$ a *sketching matrix*. For convenience in notations, we assume that the rectangular maximum volume algorithm outputs a sketching matrix. Thus, the system (5.3) can be solved as follows

$$\boldsymbol{x} = (\boldsymbol{SA})^{\dagger}(\boldsymbol{Sb}). \tag{5.5}$$

Selecting rows of $\boldsymbol{b}$ is a cheap operation, so the complexity of computing $\boldsymbol{Sb}$ is $O(P)$. If $(\boldsymbol{SA})^{\dagger}$ is precomputed, for any right-hand side we only have to carry out matrix-by-vector multiplication with a matrix of size $R \times P$.

5.2.2 Computation of Low-Dimensional Embeddings

Let $\boldsymbol{Z} \in \mathbb{R}^{N \times D}$ be the output matrix of a given layer; each row of $\boldsymbol{Z}$ corresponds to a training sample propagated through the part of the network ending with this layer.

The truncated rank-R SVD of $\boldsymbol{Z}^{\top} \in \mathbb{R}^{D \times N}$ is given by

$$\boldsymbol{Z}^{\top} \cong \underbrace{\boldsymbol{V}}_{D \times R} \underbrace{\boldsymbol{\Sigma}\boldsymbol{U}^{\top}}_{R \times N}. \tag{5.6}$$

Here the matrix $\boldsymbol{V}$ corresponds to the linear transformation, which maps to the low-dimensional embedding subspace. To compute the matrix $\boldsymbol{V}$, we use the matrix sketching algorithm based on hashing [172, 166]. For our applications, it is faster than randomized SVD.

5.3 Method

Our goal is to build an approximation of a given deep neural network (*teacher*) by another network (*student*) with much faster inference.

Most conceptual details of our approach are explained on a toy example of a multilayer perceptron (Subsection 5.3.1). Later on, we describe how to apply the proposed ideas to feed-forward convolutional neural networks (Subsection 5.3.2) and residual networks (Subsection 5.3.3).

[2] https://bitbucket.org/muxas/maxvolpy

5.3.1 A Toy Example: MLP

In this subsection, we consider a simple fully-connected feed-forward neural network, or multilayer perceptron (MLP).

Hereinafter let ψ_k ($k = 1, \ldots, K$) be non-decreasing element-wise activation functions, e.g., ReLU, ELU or Leaky ReLU. Note that our method allows us to accelerate a part of the initial network, but for simplicity, we assume that the whole teacher network is used. Besides, without loss of generality, we suppose that all biases are equal to zero.

Let $\boldsymbol{z}_0$ be an input sample. Being passed through K layers of the teacher network, it undergoes the following transformations

$$\boldsymbol{z}_1 = \psi_1(\boldsymbol{W}_1\boldsymbol{z}_0),\ \boldsymbol{z}_2 = \psi_2(\boldsymbol{W}_2\boldsymbol{z}_1),\ \ldots,\ \boldsymbol{z}_K = \boldsymbol{W}_K\boldsymbol{z}_{K-1}, \tag{5.7}$$

where $\boldsymbol{W}_k \in \mathbb{R}^{D_k \times D_{k-1}}$ is a weight matrix of the k-th layer.

Let $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$ be the embeddings of $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_K$. We have already known how to compute the linear transformation $\boldsymbol{V}_k \in \mathbb{R}^{D_k \times R_K}$, which maps $\boldsymbol{z}_k$ to $\boldsymbol{c}_k$. Here the dimensionality of the k-th embedding R_K is much smaller than the number of features D_k.

The low-rank assumption for the first layer gives

$$\boldsymbol{z}_1 \cong \boxed{\boldsymbol{V}_1\boldsymbol{c}_1 \cong \psi_1(\boldsymbol{W}_1\boldsymbol{z}_0)} \tag{5.8}$$

The boxed expression is a tall-and-skinny linear system with the matrix $\boldsymbol{V}_1 \in \mathbb{R}^{D_1 \times R_1}$, the right-hand side vector $\psi_1(\boldsymbol{W}_1\boldsymbol{z}_0)$ and the vector of unknowns $\boldsymbol{c}_1$. If $\boldsymbol{S}_1 \in \mathbb{R}^{P_1 \times D_1}$ is a sketching matrix (Section 5.2.1) for the matrix $\boldsymbol{V}_1$, we can compute the embedding as follows

$$\boldsymbol{c}_1 \cong (\boldsymbol{S}_1\boldsymbol{V}_1)^{\dagger}\,\boldsymbol{S}_1\psi_1\left(\boldsymbol{W}_1\boldsymbol{z}_0\right) = \underbrace{(\boldsymbol{S}_1\boldsymbol{V}_1)^{\dagger}}_{R_1 \times P_1}\psi_1(\underbrace{\boldsymbol{S}_1\boldsymbol{W}_1}_{P_1 \times D_1}\boldsymbol{z}_0). \tag{5.9}$$

Here we switch point-wise linearity ψ and sampling because they commute pairwise.

The same technique can be applied for computing the second embedding $\boldsymbol{c}_2$ using $\boldsymbol{c}_1$. We write the low-rank assumption

$$\boldsymbol{z}_2 \cong \psi_2\left(\boldsymbol{W}_2\boldsymbol{z}_1\right) \cong \psi_2\left(\boldsymbol{W}_2\boldsymbol{V}_1\boldsymbol{c}_1\right) \cong \boldsymbol{V}_2\boldsymbol{c}_2, \tag{5.10}$$

get the linear system

$$\boxed{\boldsymbol{V}_2\boldsymbol{c}_2 \cong \psi_2\left(\boldsymbol{W}_2\boldsymbol{V}_1\boldsymbol{c}_1\right)} \tag{5.11}$$

and apply the rectangular maximum volume algorithm. If $\boldsymbol{S}_2 \in \mathbb{R}^{P_2 \times D_2}$ is a sketching matrix, $\boldsymbol{c}_2$ can be estimated as

$$\boldsymbol{c}_2 \cong \underbrace{(\boldsymbol{S}_2\boldsymbol{V}_2)^{\dagger}}_{R_2 \times P_2}\psi_2(\underbrace{\boldsymbol{S}_2\boldsymbol{W}_2\boldsymbol{V}_1}_{P_2 \times R_1}\boldsymbol{c}_1). \tag{5.12}$$

The process can be continued for other layers. The output of the student network is computed as $\boldsymbol{V}_K \boldsymbol{c}_K$:

$$\begin{aligned}
\boldsymbol{c}_1 &\cong \underbrace{(\boldsymbol{S}_1 \boldsymbol{V}_1)^\dagger}_{R_1 \times P_1} \psi_1 (\underbrace{\boldsymbol{S}_1 \boldsymbol{W}_1}_{P_1 \times D_1} \boldsymbol{z}_0) \\
&\cdots \\
\boldsymbol{c}_k &\cong \underbrace{(\boldsymbol{S}_k \boldsymbol{V}_k)^\dagger}_{R_K \times P_k} \psi_2 (\underbrace{\boldsymbol{S}_k \boldsymbol{W}_k \boldsymbol{V}_{k-1}}_{P_k \times R_{k-1}} \boldsymbol{c}_{k-1}), \quad k = 1, \ldots, K \\
\boldsymbol{z}_K &\cong \boldsymbol{V}_K \boldsymbol{c}_K
\end{aligned} \tag{5.13}$$

Suppose $\boldsymbol{s}_k$ is the output of ψ_k. We can rewrite (5.13) in a better way

$$\begin{aligned}
\boldsymbol{s}_1 &\cong \psi_1 (\underbrace{\boldsymbol{S}_1 \boldsymbol{W}_1}_{P_1 \times D_1} \boldsymbol{z}_0), \\
\boldsymbol{s}_2 &\cong \psi_2 (\underbrace{\boldsymbol{S}_2 \boldsymbol{W}_2 \boldsymbol{V}_1 \, (\boldsymbol{S}_1 \boldsymbol{V}_1)^\dagger}_{P_2 \times P_1} \boldsymbol{s}_1), \\
&\cdots \\
\boldsymbol{s}_K &\cong \psi_K (\underbrace{\boldsymbol{S}_K \boldsymbol{W}_K \boldsymbol{V}_{K-1} \, (\boldsymbol{S}_{K-1} \boldsymbol{V}_{K-1})^\dagger}_{P_K \times P_{K-1}} \boldsymbol{s}_{K-1}), \\
\boldsymbol{z}_K &\cong \underbrace{\boldsymbol{V}_K \, (\boldsymbol{S}_K \boldsymbol{V}_K)^\dagger}_{D_k \times R_K} \boldsymbol{s}_K.
\end{aligned} \tag{5.14}$$

As a result, instead of K-layer network with $D_k \times D_{k+1}$ layers (5.7) we obtain a more compact $K+1$-layer network (5.14).

The proposed approach is summarized in Algoritm 1.

Then, we propose to add batch normalizations into the accelerated model and perform several epochs of fine-tuning.

5.3.2 Convolutional Neural Networks

Convolution is a linear transformation. We treat it as a matrix-by-vector product, and we convert convolutions to fully-connected layers. Two crucial remarks for this approach should be discussed.

Firstly, we vectorize all outputs. Do we lose the geometrical structure of the feature map? Only partially, because it is integrated into the initial weight matrices.

Secondly, the size of a single convolutional matrix is larger than the size of its kernel. However, these sizes can be compatible after compression if the number of channels is not big. So, as a result, a student model can be not only faster but even smaller than the teacher.

Batch normalization is a linear operation that uses a set of frozen weights during the inference stage. For inference, we can compose batch normalization and linear layer into a single linear layer without batch normalization. Thus,

in the student model, we get rid of batch normalization layers but preserve the normalization property.

Maximum pooling is a local operation, which typically maps 2×2 region into a single value — the maximum value in the given region. We manage this layer by taking 4 times more indices and by applying maximum pooling after sampling.

5.3.3 Residual Networks

Residual networks [73, 180, 86] are popular models used in many modern applications. In contrast to standard feed-forward CNNs, they are not sequential. Such models have several parallel branches, the outputs of which are summed up and propagated through the activation function.

We approximate the output of each branch and the result as follows

$$\boldsymbol{V}\boldsymbol{c} \cong \psi\left(\boldsymbol{V}_1\boldsymbol{c}_1 + \ldots + \boldsymbol{V}_k\boldsymbol{c}_k\right). \tag{5.15}$$

The above expression is an overdetermined linear system. If $\boldsymbol{S}$ is a sampling matrix for matrix $\boldsymbol{V}$, the embedding $\boldsymbol{c}$ is computed as

$$\boldsymbol{c} \cong \left(\boldsymbol{S}\boldsymbol{V}\right)^{\dagger} \psi\left(\boldsymbol{S}\boldsymbol{V}_1\boldsymbol{c}_1 + \ldots + \boldsymbol{S}\boldsymbol{V}_k\boldsymbol{c}_k\right). \tag{5.16}$$

The rest steps of residual network acceleration are the same as for the standard multilayer perceptron (Section 5.3.1).

5.3.4 Approximation error

Suppose $\boldsymbol{\varepsilon}_k = \boldsymbol{V}_k\boldsymbol{c}_k - \boldsymbol{z}_k$ is an error of the low-rank approximation, thus

$$\boldsymbol{S}_k\boldsymbol{V}_k\boldsymbol{c}_k = \left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\boldsymbol{S}_k\boldsymbol{z}_k + \left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\boldsymbol{S}_k\boldsymbol{\varepsilon}_k. \tag{5.17}$$

and error of our algorithm equals to $e_k = \| \left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\boldsymbol{S}_k\boldsymbol{\varepsilon}_k\|_2$. Since $\|\boldsymbol{V}_k^{\top}\|_2 = \|\boldsymbol{S}_k\|_2 = 1$,

$$\begin{aligned} \| \left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\boldsymbol{S}_k\|_2 &= \|\boldsymbol{V}_k^{\top}\boldsymbol{V}_k\left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\boldsymbol{S}_k\|_2 \\ &\leq \|\boldsymbol{V}_k\left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\|_2. \end{aligned} \tag{5.18}$$

Due to the Lemma 4.3 and Remark 4.4 from the rectangular maximum volume paper [116][3]

$$\|\boldsymbol{V}_k\left(\boldsymbol{S}_k\boldsymbol{V}_k\right)^{\dagger}\|_2 \leq \sqrt{1 + \frac{\left(D_k - P_K\right)r_k}{P_K + 1 - R_K}}. \tag{5.19}$$

Hence,

$$e_k \leq \sqrt{1 + \frac{\left(D_k - P_K\right)R_K}{P_K + 1 - R_K}}\|\varepsilon_k\|_2. \tag{5.20}$$

[3] In that paper, the given matrix is defined by C.

For example, if $P_K = 1.5R_K$, approximation error e_k is $O(\sqrt{D_k}||\varepsilon_k||_2)$ for $R_K = o(D_k)$.

5.4 Experiments

Firstly, we provide empirical evidence that supports our low-rank assumption about the outputs of some layers. Secondly, we show the performance of RON for both fully-connected and convolutional neural network architectures. We compare models accelerated by RON with the baselines from DCP [186] and [184].

Datasets. We empirically evaluate the performance of RON on four datasets, including MNIST, CIFAR-10, CIFAR-100, and SVHN. MNIST is a collection of handwritten digits that contains images of size 28×28 with a training set of 60000 examples, and a test set of 10000 examples. CIFAR-10 consists of 32×32 color images belonging to 10 classes with 50000 training and 10000 testing samples. CIFAR-100 is similar to CIFAR-10, but it contains 100 classes with 500 training images and 100 testing images per class. SVHN is a real-world image dataset with house numbers that contains 73257 training and 26032 testing images of size 32×32.

5.4.1 Singular values

Our method relies on the assumption, which states that the outputs of some layers can be mapped to a low-dimensional space. We perform this mapping using the maximum volume based approximation of the basis obtained through SVD. Figure 5.1 supports the feasibility of our assumption. Each subfigure corresponds to a specific architecture and depicts the singular values of blocks output matrices. It can be seen that the singular values decrease very fast for some (deeper) blocks, which means that their outputs can be approximated by low-dimensional embeddings.

We use two strategies for rank selection: a non-parametric Variational Bayesian Matrix Factorization (VBMF, [121]) and a simple constant factor rank reduction.

Singular values are computed for matrices containing the whole training data. We use matrix sketching algorithm based on hashing and do not have to store the entire matrix in memory [172, 166].

Since the error is propagated through the layers and can be accumulated, we fine-tune the processed model.

5.4.2 Fully-connected networks

To illustrate our method, we first choose LeNet-300-100 architecture for MNIST, which is a fully-connected networks with three layers: 784×300, 300×100, and 100×10 with ReLU activations. We perform 15 iterations of the following procedure. First, we train the model with the learning rate `1e-3` for 25 epochs, then we train the model with the learning rate `5e-4` for the

Algorithm 1: Initialization of the student network

Input: teacher's weights $\{\boldsymbol{W}_1, \boldsymbol{W}_2, \ldots, \boldsymbol{W}_K\}$ and element-wise activation functions $\{\psi_1, \psi_2, \ldots, \psi_K\}$; subset of the training set $\boldsymbol{Z}$ — a number of samples $\times$ number of input features matrix; $\{R_1, R_2, \ldots, R_K\}$ — sizes of the embeddings;

Output: student's weights $\{\widetilde{\boldsymbol{W}}_0, \widetilde{\boldsymbol{W}}_1, \widetilde{\boldsymbol{W}}_2, \ldots, \widetilde{\boldsymbol{W}}_K\}$;

`/* For simplicity, we use all` $\{\boldsymbol{V}_k\}_{k=1}^{K}$`, but in fact we have to keep only two of them to compute a single weight of student. */`

for $k \leftarrow 1$ ***to*** K **do**
  $\boldsymbol{Z} \leftarrow \boldsymbol{Z}$ propagated through the k-th layer
  $\boldsymbol{U}, \boldsymbol{\Sigma}, \boldsymbol{V}_k \leftarrow \text{truncated\_svd}(\boldsymbol{Z}^\top, R_K)$
  `/* In practice, we don't store the whole` $\boldsymbol{Z}$`, but use streaming randomized SVD algorithms */`
  $\boldsymbol{S}_k \leftarrow \text{rect\_max\_vol}(\boldsymbol{V}_k)$
end
$\widetilde{\boldsymbol{W}}_0 \leftarrow \boldsymbol{S}_1 \boldsymbol{W}_1$
for $k \leftarrow 1$ ***to*** $K - 1$ **do**
  $\widetilde{\boldsymbol{W}}_k \leftarrow \boldsymbol{S}_k \boldsymbol{W}_k \boldsymbol{V}_{k-1} \left(\boldsymbol{S}_{k-1} \boldsymbol{V}_{k-1}\right)^\dagger$
end
$\widetilde{\boldsymbol{W}}_K \leftarrow \boldsymbol{V}_K \left(\boldsymbol{S}_K \boldsymbol{V}_K\right)^\dagger$
return $\{\widetilde{\boldsymbol{W}}_0, \widetilde{\boldsymbol{W}}_1, \widetilde{\boldsymbol{W}}_2, \ldots, \widetilde{\boldsymbol{W}}_K\}$

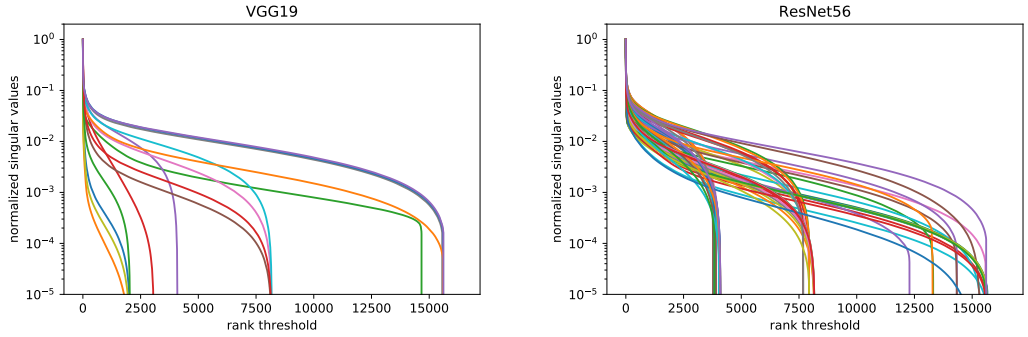
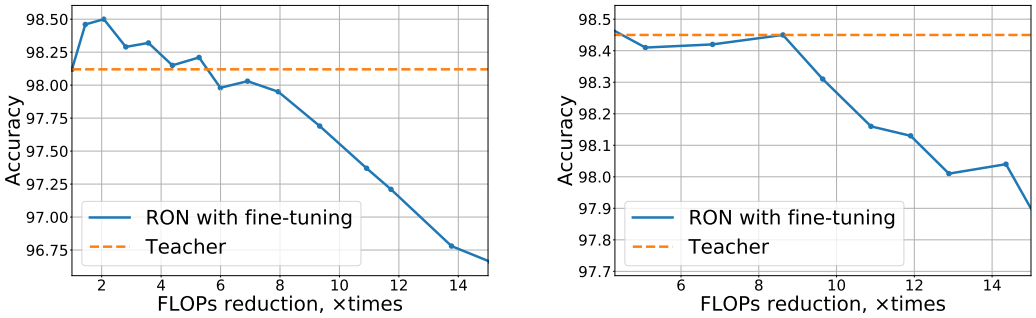


FIGURE 5.1: We plot singular values of all layers for CIFAR-10 for VGG-19 (left) and ResNet-56 (right). Each singular value is divided by the largest one for this layer. One can see that most singular values are relatively small.

same number of epochs. After that, we apply our acceleration procedure with rank reduction rates equal to 0.7 and 0.75, respectively. Figure 5.2a shows the FLOP reduction rate together with the test accuracy.



(a) LeNet-300-100-10 (b) LeNet-500-300-10

FIGURE 5.2: RON for different LeNet models.

In [170, 107, 184], LeNet-500-300 model is accelerated 6.06, 6.41, and 7.85 times, respectively, approximately without accuracy drop. Our method is able to achieve more then 8× acceleration (see Figure 5.2b).

5.4.3 Convolutional networks

We apply our method to VGG-like [159] architectures on CIFAR-10, CIFAR-100, and SVHN classification tasks. In our experiments, we use RON once to obtain an accelerated neural network (student), and then we fine-tune the network if needed. During student initialization (Algorithm 1), embedding sizes for CIFAR-10 are chosen using VBMF, while for CIFAR-100 and SVHN feature sizes are reduced by a predefined rate. Pre-trained teacher models for CIFAR-10 are available online[4]. They include VGG-19 and VGG-19 pruned with DCP [186] approach at 0.3% pruning rate. For the experiments with

[4] https://github.com/SCUT-AILab/DCP/wiki/Model-Zoo

CIFAR-100 and SVHN we used pre-trained VGG-19[5] and VGG-7[6] networks, correspondingly. When applying RON to VGG-like architecture, we accelerate several last convolutional layers of the network. For instance, the model RON (8 to 16) corresponds to the model with nine accelerated convolutional layers.

RON without fine-tuning. In this setting, we initialize a student network using Algorithm 1 and measure its acceleration and performance. For VGG-19 on CIFAR-10, RON can achieve 1.53× FLOP reduction with 0.09% accuracy increase without any additional fine-tuning (Table 1). We refer to [184] to provide evidence that RON (without fine-tuning) significantly outperforms one stage of channel pruning [107, 184](w/o fine-tuning) when applied to the pre-trained VGG-19. The models with convolutional layers pruned at 0.1% pruning rate (i.e., FLOP reduction is around 1.23×) have more than 20% accuracy drop (Figure 5 in [184]).

RON with fine-tuning. Fine-tuning the model accelerated with RON, we can achieve 2.3× FLOP reduction with 0.28% accuracy increase (Table 5.1) for VGG-19 on CIFAR-10. After the acceleration procedure, we perform 250 epochs of fine-tuning by SGD with momentum 0.9, weight decay `1e-4`, and batch size 256. The initial learning rate is equal to `1e-2`, and it is halved after ten training epochs without validation quality improvement. We use dropout during the fine-tuning.

VGG networks for CIFAR-100 (Table 5.2) and SVHN (Table 5.3) are less redundant, therefore, acceleration without accuracy drop is smaller than for CIFAR-10 dataset.

Note that compression can be performed iteratively by alternating RON and fine-tuning steps. The iterative approach takes much time, but it was shown for both pruning [107, 186, 50, 184] and low-rank [64] methods that it helps to reduce the accuracy degradation for high compression ratios.

RON on top of the pruned network. The motivation to use RON on top of pruned models is the following. Channel pruning methods tend to leave the most informative channels (e.g., in DCP [186] they look for more discriminative channels) and eliminate the rest. However, a convolutional layer consisting of informative channels can still have a low-rank structure and, therefore, can be further accelerated using RON. For VGG-19 pruned with DCP [186] approach at 0.3% pruning rate, RON provides 4.48× FLOP reduction with 0.27% accuracy increase comparing to the initial VGG-19 while maintaining higher accuracy and better acceleration than the pruned baseline (Figure 5.3).

5.4.4 Comparisons with other approaches

The advantage of our method is that it can be applied both alone and on top of pruning algorithms. We have aggregated our results (Table 5.1) with

[5] `https://github.com/bearpaw/pytorch-classification`
[6] `https://github.com/aaron-xichen/pytorch-playground`

TABLE 5.1: Accuracy and FLOP trade-off for the models accelerated with RON on CIFAR-10 dataset. DCP is a channel pruning method from [186].

Model	Modified layers	Acc@1 without fine-tuning	Acc@1 with fine-tuning	FLOP reduction
Teacher	—	—	93.70	1.00×
RON	10 to 16	**93.79**	**94.10**	**1.53**×
RON	9 to 16	93.46	**94.15**	**1.68**×
RON	8 to 16	90.58	**94.24**	**1.93**×
RON	7 to 16	85.79	**93.98**	**2.30**×
RON	6 to 16	72.53	93.12	3.01×
RON	5 to 16	58.12	91.88	3.66×
DCP [186]	—	—	93.96	2.00×
DCP + RON	10 to 16	**93.98**	**94.24**	**3.06**×
DCP + RON	9 to 16	**93.90**	**94.27**	**3.37**×
DCP + RON	8 to 16	91.82	**94.01**	**3.78**×
DCP + RON	7 to 16	88.88	**93.97**	**4.48**×
DCP + RON	6 to 16	81.30	93.26	5.56×
DCP + RON	5 to 16	64.12	91.5	7.21×

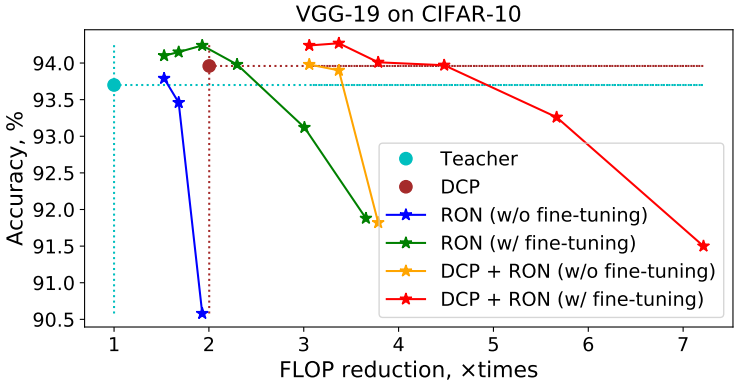


FIGURE 5.3: Accuracy and FLOP reduction for RON accelerated models on CIFAR-10.

the information from the paper by Zhuang et al. [186] and present them in Table 5.4. We compare RON with ThiNet [111], channel pruning (CP) [75], network slimming [107] and width-multiplier method [49]. More details about related methods can be found in Section 5.6.

5.5 Discussion

We have proposed a method that exploits the low-rank property of the outputs of neural network layers. The advantage of our approach is the ability to work with a large class of modern neural networks and obtain a simple fully-connected student neural network. We showed that, in some cases, the

TABLE 5.2: VGG on CIFAR-100. RON $N\times$ stands for the accelerated model, where feature dimentionality of last layers is reduced by $N\times$ times comparing to the teacher.

Model	Modified layers	Acc@1 without fine-tuning	Acc@5 without fine-tuning	Acc@1 with fine-tuning	Acc@5 with fine-tuning	Speed up on CPU	FLOP reduction
Teacher	—	—	—	71.95	89.41	1.00×	1.00×
RON 10×	8 to 16	70.81	88.51	**72.09**	**90.12**	**1.95**×	**1.66** ×
RON 20×	8 to 16	63.94	85.12	71.89	89.95	2.15×	1.71×
RON 10×	10 to 16	60.68	82.36	70.87	90.46	1.72×	1.84×
RON 20×	10 to 16	44.07	68.29	69.69	89.78	2.19×	2.19×
RON 10×	12 to 16	42.77	67.34	66.84	88.16	2.22×	2.58×

TABLE 5.3: VGG on SVHN. RON $N\times$ stands for the accelerated model, where feature dimentionality of last layers is reduced by $N\times$ times comparing to the teacher.

Model	Modified layers	Acc@1 without fine-tuning	Acc@1 with fine-tuning	Speed up on CPU	FLOP reduction
Teacher	—	—	96.03	1.00×	1.00×
RON 10×	5 to 7	92.46	95.41	1.62×	1.30×
RON 20×	5 to 7	89.04	95.33	1.71×	1.53×
RON 20×	3 to 7	83.58	92.13	1.67×	1.65×

TABLE 5.4: Comparison of acceleration methods for VGG-19 on CIFAR-10. Pre-trained baseline has 93.7% accuracy. The higher FLOP reduction the better. The smaller accuracy drop the better.

Model	FLOP reduction	Accuracy drop, %
ThiNet [111]	2.00×	0.14
Network Sliming [107]	2.04×	0.19
Channel Pruning [75]	2.00×	0.32
Width-multiplier [49]	2.00×	0.38
Discrimination-aware Channel Pruning (DCP) [186]	2.00×	-0.17
DCP-Adapt [186]	2.86×	-0.58
RON (modified layers: 7 to 16) + fine-tuning	**2.30**×	**-0.18**
DCP + RON (modified layers: 9 to 16) + fine-tuning	**3.37**×	**-0.57**
DCP + RON (modified layers: 7 to 16) + fine-tuning	**4.48**×	**-0.27**

student model has the same quality as a student network even without any fine-tuning.

The disadvantage of the Reduced-Order Network is that the number of parameters may increase when applied to wide convolutional networks on high-resolution images since the resulting network is dense. However, we have demonstrated that our method works well for neural networks with pruned channels, and such pruning allows us to reduce the number of features. The best application of our approach, in our opinion, is to further accelerate networks, which were produced by channel pruning algorithms.

Later on, we can try sparsification [118] and quantization techniques on top of our approach to mitigate this issue.

5.6 Related work

Recently, a series of approaches have been proposed to speed up inference in convolutional neural networks (CNNs) [23]. In this section, we overview the main ideas of different method families and highlight the differences between them and our approach.

Many different methods deal with a pre-trained network, which we call the *teacher* network, and an accelerated network, which we call the *student* network. This terminology comes from **knowledge distillation** [14, 79, 141, 179] methods, where the softmax outputs of the teacher network are used as a target vector for the student network.

In section 5.4.4, we compare our aproach with different **channel pruning** methods. Such methods aim to prune redundant channels in the weight tensors and, hence, accelerate and compress the whole model. Pruned channels are selected due to special information criteria. For example, it can be a sum of absolute values of weights [102] or an average percentage of zeros [85].

There are two major approaches to channel pruning. The first approach is to deal with a single network and train it from scratch, adding extra regularization, which forces the channel-level sparsity of weights. Later on, some channels are considered to be redundant and have to be removed [107, 170]. It is usually an iterative procedure, which is computationally expensive, especially for very deep neural networks.

The student is trained to minimize the reconstruction error between feature maps of two models [75, 85, 111].

In [75], channel selection is made using LASSO regression, and the reconstruction is performed via least squares. In [111], the pruning strategy for a layer depends on the statistics of the next layer. In [107], it is proposed to multiply each channel on a unique learnable scalar parameter; then, the whole network is trained with a sparsity regularization on these scalar parameters. In [184], neural architecture search techniques are combined with channel pruning. Namely, the pruning strategy is generated by the LSTM network, which is trained in a reinforcement learning way.

In [186], Discrimination-aware Channel Pruning (DCP) algorithm is introduced. It is a multi-stage pruning method applied to the pre-trained network. At each stage of DCP, a network from the previous stage is trained with an additional classifier and discriminative loss. The least informative channels either pruned at a fixed rate or selected using a greedy algorithm. We refer to these approaches as to DCP and DCP-Adapt, accordingly.

Another related family of acceleration methods is **low-rank methods**, which uses matrix or tensor decomposition to estimate the informative parameters of deep neural networks. In most cases, a much lower total computational cost can be achieved by replacing a convolutional layer with a sequence of several smaller convolutional layers [39, 89, 101, 183, 64]. Opposed to our

approach, most low-rank methods are applied not to feature maps [35] but to weight tensors.

Finally, **quantization** [30, 63] methods worth mentioning. Such methods can significantly accelerate networks, but they usually require special hardware to reach a theoretical speed-up in practice.

5.7 Conclusion

We have developed a neural network inference acceleration method that is based on mapping layer outputs to a low-dimensional subspace using the singular value decomposition and the rectangular maximum volume algorithm. We demonstrated empirically that our approach allows finding a good initial approximation in the space of new model parameters. Namely, on CIFAR-10 and CIFAR-100, we achieved accuracy on par or even slightly better than the teacher model without fine-tuning and reached acceleration up to 4.48$\times$ with fine-tuning and no accuracy drop. We have supported our experiments with the theoretical results, including approximation error upper bound evaluation.

Chapter 6

Active Subspaces for Neural Networks

6.1 Introduction

Deep neural networks have demonstrated impressive performance in a range of applications, such as computer vision [99], natural language processing [177], and speech recognition [62]. These networks often utilize deep structures with many layers and a large number of neurons to achieve high accuracy and expressive power [127, 52]. However, it is not always clear how many layers and neurons are necessary, and using an unnecessarily complex deep neural network can result in increased runtime and hardware requirements. As a result, there is growing interest in constructing smaller neural networks for resource-constrained applications, such as robotics and the internet of things, by removing network redundancy. Representative methods for reducing the size of neural networks include network pruning and sharing [48, 70, 76, 108, 105], low-rank matrix and tensor factorization [147, 72, 51, 101, 124], parameter quantization [31, 38], and knowledge distillation [80, 141], among others. However, most existing methods delete model parameters directly without changing the network architecture [76, 70, 16, 105].

Another important issue of deep neural networks is the lack of robustness. Robustness in deep neural networks (DNNs), as DNNs are often used in safety-critical applications such as autonomous driving and medical image analysis, and are expected to perform well even when faced with noisy or corrupted data. However, studies have shown that many state-of-the-art DNNs are vulnerable to small perturbations [164]. A variety of methods have been proposed to generate adversarial examples, including optimization methods [17, 119, 120, 164], sensitive features [59, 129], geometric transformations [42, 91], and generative models [8, 153]. However, these methods have the limitation of requiring the computation of a new perturbation for each new data sample. Recently, several methods have been proposed to compute a universal adversarial attack that can fool a dataset as a whole in various applications, such as computer vision[120], speech recognition [122], audio [1], and text classification [9]. However, these methods still rely on solving a series of data-dependent sub-problems. In [93], Khrulkov et al. proposed a method for constructing universal perturbations by computing the (p, q)-singular vectors of the Jacobian matrices of hidden layers in a network.

This chapter investigates the above two issues with the active subspace method [145, 25, 27] that was originally developed for uncertainty quantification. The key idea of the active subspace is to identify the low-dimensional subspace constructed by some important directions that can contribute significantly to the variance of the multi-variable function. These directions are corresponding to the principal components of the uncentered covariance matrix of gradients. Afterwards, a response surface can be constructed in this low-dimensional subspace to reduce the number of parameters for partial differential equations [27] and uncertainty quantification [28]. However, the power of active subspace in analyzing and attacking deep neural networks has not been explored.

6.1.1 Contributions

The contribution of this chapter is twofold.

- Firstly, according to Figure 6.1 (a), we find that only a small number of neurons can be important when applying the active subspace to some intermediate layers of a deep neural network. This motivates us to define the concept of "active neurons". Figure 6.1 (b) also shows that most parameters are concentrated in the final layers. Therefore, we propose ASNet, a simpler framework that removes the final layers and replaces them with an active-subspace layer that maps intermediate neurons to a low-dimensional subspace and a polynomial chaos expansion layer. Our numerical experiments show that ASNet has fewer parameters than the original network. ASNet can also be combined with structured re-training methods to achieve better accuracy with fewer parameters.

- Secondly, we use active subspace to develop a new universal attack method to fool deep neural networks on a whole data set. We formulate this problem as a ball-constrained loss maximization problem and propose a heuristic projected gradient descent algorithm to solve it. At each iteration, the ascent direction is the dominant active subspace, and the stepsize is decided by the backtracking algorithm. Figure 6.1 (c) shows that the attack ratio of the active subspace direction is much higher than that of the random vector.

The rest of this chapter is organized as follows. In Section 6.2, we review the key idea of active subspace. Based on the active-subspace method, Section 6.3 shows how to find the number of active neurons in a deep neural network and further proposes a new and compact network, referred to as ASNet. Section 6.4 develops a new universal adversarial attack method based on active subspace. The numerical experiments for both ASNet and universal adversarial attacks are presented in Section 6.5. Finally, we conclude this chapter in Section 6.6.

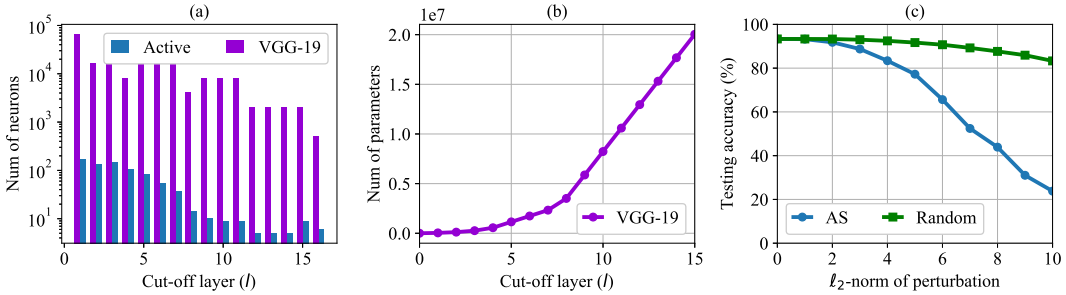


FIGURE 6.1: Structural analysis of deep neural networks by the active subspace (AS). All experiments are conducted on CIFAR-10 by VGG-19. (a) The number of neurons can be significantly reduced by the active subspace. Here, the number of active neurons is defined by Definition 6.3.1 with a threshold $\epsilon = 0.05$; (b) Most of the parameters are distributed in the last few layers; (c) The active subspace direction can perturb the network significantly.

6.2 Active Subspace

Active subspace is an efficient tool for functional analysis and dimension reduction. Its key idea is to construct a low-dimensional subspace for the input variables in which the function value changes dramatically. Given a continuous function $c(\mathbf{x})$ with $\mathbf{x}$ described by the probability density function $\rho(\mathbf{x})$, one can construct an uncentered covariance matrix for the gradient: $\mathbf{C} = \mathbb{E}[\nabla c(\mathbf{x}) \nabla c(\mathbf{x})^T]$. Suppose the matrix $\mathbf{C}$ admits the following eigenvalue decomposition,

$$\mathbf{C} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^T, \tag{6.1}$$

where $\mathbf{V}$ includes all orthogonal eigenvectors and

$$\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_n), \ \lambda_1 \geq \cdots \geq \lambda_n \geq 0 \tag{6.2}$$

are the eigenvalues. All the eigenvalues are nonnegative because $\mathbf{C}$ is positive semidefinite. One can split the matrix $\mathbf{V}$ into two parts,

$$\mathbf{V} = [\mathbf{V}_1, \ \mathbf{V}_2], \text{ where } \mathbf{V}_1 \in \mathbb{R}^{n \times r} \text{ and } \mathbf{V}_2 \in \mathbb{R}^{n \times (n-r)}. \tag{6.3}$$

The subspace spanned by matrix $\mathbf{V}_1 \in \mathbb{R}^{n \times r}$ is called an active subspace [145] because $c(\mathbf{x})$ is sensitive to perturbation vectors within this subspace.

Remark 1 (Relationships with the Principal Component Analysis) *Given a set of data* $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^m]$ *with each column representing a data sample and each row being zero-mean, the first principal component* $\mathbf{w}_1$ *inherits the maximal variance from* $\mathbf{X}$*, namely,*

$$\mathbf{w}_1 = \underset{\|\mathbf{w}\|_2 = 1}{argmax} \sum_{i=1}^{m} (\mathbf{w}_1^T \mathbf{x}^i)^2 = \underset{\|\mathbf{w}\|_2 = 1}{argmax} \ \mathbf{w}^T \mathbf{X} \mathbf{X}^T \mathbf{w}. \tag{6.4}$$

The variance is maximized when $\mathbf{w}_1$ *is the eigenvector associated with the largest eigenvalue of* $\mathbf{X}\mathbf{X}^T$. *The first r principal components are the r eigenvectors associated with the r largest eigenvalues of* $\mathbf{X}\mathbf{X}^T$. *The main difference with the active subspace is that the principal component analysis uses the covariance matrix of input data sets* $\mathbf{X}$, *but the active-subspace method uses the covariance matrix of gradient* $\nabla c(\mathbf{x})$. *Hence, a perturbation along the direction* $\mathbf{w}_1$ *from (6.4) only guarantees the variability in the data, and does not necessarily cause a significant change in the value of* $c(\mathbf{x})$.

The following lemma quantitatively describes that $c(\mathbf{x})$ varies more on average along the directions defined by the columns of $\mathbf{V}_1$ than the directions defined by the columns of $\mathbf{V}_2$.

Lemma 6.2.1 *[27] Suppose* $c(\mathbf{x})$ *is a continuous function and* $\mathbf{C}$ *is obtained from (6.1). For the matrices* $\mathbf{V}_1$ *and* $\mathbf{V}_2$ *generated by (6.3), and the reduced vector*

$$\mathbf{z} = \mathbf{V}_1^T\mathbf{x} \text{ and } \tilde{\mathbf{z}} = \mathbf{V}_2^T\mathbf{x}, \tag{6.5}$$

it holds that

$$\begin{aligned}\mathbb{E}_{\mathbf{x}}[\nabla_{\mathbf{z}} c(\mathbf{x})^T \nabla_{\mathbf{z}} c(\mathbf{x})] =& \lambda_1 + \ldots + \lambda_r,\\ \mathbb{E}_{\mathbf{x}}[\nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})^T \nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})] =& \lambda_{r+1} + \ldots + \lambda_n.\end{aligned} \tag{6.6}$$

Sketch of proof [27]:

$$\begin{aligned}&\mathbb{E}_{\mathbf{x}}[\nabla_{\mathbf{z}} c(\mathbf{x})^T \nabla_{\mathbf{z}} c(\mathbf{x})]\\ =&\text{trace}\left(\mathbb{E}_{\mathbf{x}}[\nabla_{\mathbf{z}} c(\mathbf{x}) \nabla_{\mathbf{z}} c(\mathbf{x})^T]\right)\\ =&\text{trace}\left(\mathbb{E}_{\mathbf{x}}[\mathbf{V}_1^T \nabla_{\mathbf{x}} c(\mathbf{x}) \nabla_{\mathbf{x}} c(\mathbf{x})^T \mathbf{V}_1]\right)\\ =&\text{trace}\left(\mathbf{V}_1^T \mathbf{C} \mathbf{V}_1\right)\\ =&\lambda_1 + \ldots + \lambda_r.\end{aligned}$$

When $\lambda_{r+1} = \ldots = \lambda_n = 0$, Lemma 6.2.1 implies $\nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})$ is zero everywhere, i.e., $c(\mathbf{x})$ is $\tilde{z}$-invariant. In this case, we may reduce $\mathbf{x} \in \mathbb{R}^n$ to a low-dimensional vector $\mathbf{z} = \mathbf{V}_1^T\mathbf{x} \in \mathbb{R}^r$ and construct a new response surface $g(\mathbf{z})$ to represent $c(\mathbf{x})$. Otherwise, if λ_{r+1} is small, we may still construct a response surface $g(\mathbf{z})$ to approximate $c(\mathbf{x})$ with a bounded error, as shown in the following lemma.

6.2.1 Response Surface

For a fixed $\mathbf{z}$, the best guess for g is the conditional expectation of c given $\mathbf{z}$, i.e.,

$$g(\mathbf{z}) = \mathbb{E}_{\tilde{\mathbf{z}}}[c(\mathbf{x})|\mathbf{z}] = \int c(\mathbf{V}_1\mathbf{z} + \mathbf{V}_2\tilde{\mathbf{z}})\rho(\tilde{\mathbf{z}}|\mathbf{z})d\tilde{\mathbf{z}}. \tag{6.7}$$

Based on the Poincaré inequality, the following approximation error bound is obtained [27].

Lemma 6.2.2 *Assume that $c(\mathbf{x})$ is absolutely continuous and square integrable with respect to the probability density function $\rho(\mathbf{x})$, then the approximation function $g(\mathbf{z})$ in (6.7) satisfies:*

$$\mathbb{E}[(c(\mathbf{x}) - g(\mathbf{z}))^2] \leq O(\lambda_{r+1} + \ldots + \lambda_n). \tag{6.8}$$

The sketch of proof from [27]:

$$\begin{aligned}
&\mathbb{E}_{\mathbf{x}}[(c(\mathbf{x}) - g(\mathbf{z}))^2] \\
=&\mathbb{E}_{\mathbf{z}}[\mathbb{E}_{\tilde{\mathbf{z}}}[(c(\mathbf{x}) - g(\mathbf{z}))^2 \,|\mathbf{z}]] \\
\leq&\text{const} \times \mathbb{E}_{\mathbf{z}}[\mathbb{E}_{\tilde{\mathbf{z}}}[\nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})^T \nabla_{\tilde{\mathbf{z}}} c(\mathbf{x}) | \mathbf{z}]] \quad \text{(Poincaré inequality)} \\
=&\text{const} \times \mathbb{E}_{\mathbf{x}}[\nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})^T \nabla_{\tilde{\mathbf{z}}} c(\mathbf{x})] \\
=&\text{const} \times (\lambda_{r+1} + \ldots + \lambda_n) \quad \text{(Lemma 6.2.1)} \\
=&O(\lambda_{r+1} + \ldots + \lambda_n).
\end{aligned}$$

In other words, the active-subspace approximation error will be small if $\lambda_{r+1}, \ldots, \lambda_n$ are negligible.

6.3 Active Subspace for Structural Analysis and Compression of Deep Neural Networks

This section applies the active subspace to analyze the internal layers of a deep neural network to reveal the number of important neurons at each layer. Afterward, a new network called ASNet was built to reduce the storage and computational complexity.

6.3.1 Deep Neural Networks

Many deep learning architectures can be described as

$$f(\mathbf{x}_0) = f_L\left(f_{L-1} \ldots (f_1(\mathbf{x}_0))\right), \tag{6.9}$$

where $\mathbf{x}_0 \in \mathbb{R}^{n_0}$ is an input, L is the total number of layers, and $f_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ is a function representing the l-th layer (e.g., combinations of convolution or fully connected, batch normalization, ReLU, pooling layers, and skip-connections). For any $1 \leq l \leq L$, we rewrite the above feed-forward model as a superposition of functions, i.e.,

$$f(\mathbf{x}_0) = f^l_{\text{post}}(f^l_{\text{pre}}(\mathbf{x}_0)), \tag{6.10}$$

where the **pre-model** $f^l_{\text{pre}}(\cdot) = f_l \ldots (f_1(\cdot))$ denotes all operations before the l-th layer and the **post-model** $f^l_{\text{post}}(\cdot) = f_L \ldots (f_{l+1}(\cdot))$ denotes all succeeding operations. The intermediate neuron $\mathbf{x}_l = f^l_{\text{pre}}(\mathbf{x}_0) \in \mathbb{R}^{n_l}$ usually lies in a high dimension. We aim to study whether such a high dimensionality is necessary. If not, how can we reduce it?

6.3.2 The Number of Active Neurons

Denote loss$(\cdot)$ as the loss function, and

$$c_l(\mathbf{x}) = \text{loss}(f^l_{\text{post}}(\mathbf{x})). \tag{6.11}$$

The covariance matrix $\mathbf{C} = \mathbb{E}[\nabla c_l(\mathbf{x}) \nabla c_l(\mathbf{x})^\top]$ admits the eigenvalue decomposition $\mathbf{C} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$ with $\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_{n_l})$. We try to extract the active subspace of $c_l(\mathbf{x})$ and reduce the intermediate vector $\mathbf{x}$ to a low dimension. Here the intermediate neuron $\mathbf{x}$, the covariance matrix $\mathbf{C}$, eigenvalues $\mathbf{\Lambda}$, and eigenvectors $\mathbf{V}$ are also related to the layer index l, but we ignore the index for simplicity.

Definition 6.3.1 *Suppose $\mathbf{\Lambda}$ is computed by (6.2). For any layer index $1 \le l \le L$, we define **the number of active neurons** $n_{l,AS}$ as follows:*

$$n_{l,AS} = \arg\min \left\{ i : \frac{\lambda_1 + \ldots + \lambda_i}{\lambda_1 + \ldots + \lambda_{n_l}} \ge 1 - \epsilon \right\}, \tag{6.12}$$

where $\epsilon > 0$ is a user-defined threshold.

Based on Definition 6.3.1, the post-model can be approximated by an $n_{l,\text{AS}}$-dimensional function with a high accuracy, i.e.,

$$g_l(\mathbf{z}) = \mathbb{E}_{\tilde{\mathbf{z}}}[c_l(\mathbf{x})|\mathbf{z}]. \tag{6.13}$$

Here $\mathbf{z} = \mathbf{V}_1^\top \mathbf{x} \in \mathbb{R}^{n_{l,AS}}$ plays the role of active neurons, $\tilde{\mathbf{z}} = \mathbf{V}_2^\top \mathbf{x} \in \mathbb{R}^{n - n_{l,AS}}$, and $\mathbf{V} = [\mathbf{V}_1, \mathbf{V}_2]$.

Lemma 6.3.1 *Suppose the input $\mathbf{x}_0$ is bounded. Consider a deep neural network with the following operations: convolution, fully connected, ReLU, batch normalization, max-pooling, and equipped with the cross entropy loss function. Then for any $l \in \{1, \ldots, L\}$, $\mathbf{x} = f^l_{pre}(\mathbf{x}_0)$, and $c_l(\mathbf{x}) = loss(f^l_{post}(\mathbf{x}))$, the $n_{l,AS}$-dimensional function $g_l(\mathbf{z})$ defined in (6.13) satisfies*

$$\mathbb{E}_{\mathbf{z}} \left[(g_l(\mathbf{z}))^2 \right] \le 2\mathbb{E}_{\mathbf{x}_0} \left[(c_0(\mathbf{x_0}))^2 \right] + O(\epsilon). \tag{6.14}$$

Denote $c_l(\mathbf{x}) = \text{loss}(f_L(\ldots(f_{l+1}(\mathbf{x}))))$, where $\text{loss}(\mathbf{y}) = -\log \frac{\exp(y_b)}{\sum_{i=1}^{n_L} \exp(y_i)}$ is the cross entropy loss function, b is the true label, and n_L is the total number of classes. We first show $c_l(\mathbf{x})$ is absolutely continuous and square-integrable, and then apply Lemma 6.2.2 to derive (6.14).

Firstly, all components of $c_l(\mathbf{x})$ are Lipschitz continuous because (1) the convolution, fully connected, and batch normalization operations are all linear; (2) the max pooling and ReLU functions are non-expansive. Here, a mapping m is non-expansive if $\|m(\mathbf{x}) - m(\mathbf{y})\| \le \|\mathbf{x} - \mathbf{y}\|$; (3) the cross entropy loss function is smooth with an upper bounded gradient, i.e., $\|\nabla \text{loss}(\mathbf{y})\| = \|\mathbf{e}_b - \exp(\mathbf{y}) / \sum_{i=1}^{n_L} \exp(y_i)\| \le \sqrt{n_L}$. The composition of two Lipschitz continuous functions is also Lipschitz continuous: suppose the Lipschitz constants for

f_1 and f_2 are α_1 and α_2, respectively, it holds that $\|f_1(f_2(\bar{\mathbf{x}})) - f_1(f_2(\underline{\mathbf{x}}))\| \leq \alpha_1\|f_2(\bar{\mathbf{x}}) - f_2(\underline{\mathbf{x}})\| \leq \alpha_1\alpha_2\|\bar{\mathbf{x}} - \underline{\mathbf{x}}\|$ for any vectors $\bar{\mathbf{x}}$ and $\underline{\mathbf{x}}$. By recursively applying the above rule, $c_l(\mathbf{x})$ is Lipschitz continuous:

$$\begin{aligned}\|c_l(\bar{\mathbf{x}}) - c_l(\underline{\mathbf{x}})\|_2 &= \|\text{loss}(f_L(\ldots(f_{l+1}(\bar{\mathbf{x}})))) - \text{loss}(f_L(\ldots(f_{l+1}(\underline{\mathbf{x}}))))\|_2 \\ &\leq \sqrt{n_L}\alpha_L \ldots \alpha_{l+1}\|\bar{\mathbf{x}} - \underline{\mathbf{x}}\|_2.\end{aligned}$$

The intermediate neuron $\mathbf{x}$ is in a bounded domain because the input $\mathbf{x}_0$ is bounded, and all functions $f_i(\cdot)$ are either continuous or non-expansive. Based on the fact that any Lipschitz-continuous function is also absolutely continuous on a compact domain [143], we conclude that $c_l(\mathbf{x})$ is absolutely continuous.

Secondly, because $\mathbf{x}$ is bounded and $c_l(\mathbf{x})$ is continuous, both $c_l(\mathbf{x})$ and its square integral will be bounded, i.e., $\int (c_l(\mathbf{x})^2\rho(\mathbf{x})d\mathbf{x} < \infty$.

Finally, by Lemma 6.2.2, it holds that

$$\mathbb{E}_{\mathbf{x}}[(c_l(\mathbf{x}) - g_l(\mathbf{z}))^2] \leq O(\lambda_{n_{l,AS}+1} + \ldots + \lambda_n).$$

From Definition 6.3.1, we have

$$\lambda_{n_{l,AS}+1} + \ldots + \lambda_n \leq (\lambda_1 + \ldots + \lambda_n)\epsilon = \|\mathbf{C}^{1/2}\|_F^2\epsilon = O(\epsilon).$$

In the last equality, we used that $\|\mathbf{C}^{1/2}\|_F$ is upper bounded because $c_l(\mathbf{x})$ is Lipschitz continuous with a bounded gradient. Consequently, we have

$$\begin{aligned}&\mathbb{E}_{\mathbf{x}}[(g_l(\mathbf{z}))^2] \\ =&\mathbb{E}_{\mathbf{x}}[(g_l(\mathbf{z}) - c_l(\mathbf{x}) + c_l(\mathbf{x}))^2] \\ \leq&2\mathbb{E}_{\mathbf{x}}[(c_l(\mathbf{x}))^2] + 2\mathbb{E}_{\mathbf{x}}[(c_l(\mathbf{x}) - g_l(\mathbf{z}))^2] \\ =&2\mathbb{E}_{\mathbf{x}_0}[(c_0(\mathbf{x_0}))^2] + 2\mathbb{E}_{\mathbf{x}}[(c_l(\mathbf{x}) - g_l(\mathbf{z}))^2] \\ \leq&2\mathbb{E}_{\mathbf{x}_0}[(c_0(\mathbf{x_0}))^2] + O(\epsilon).\end{aligned}$$

The proof is completed.

The above lemma shows that the active subspace method can reduce the number of neurons of the l-th layer from n_l to $n_{l,AS}$. The loss for the low-dimensional function $g_l(\mathbf{z})$ is bounded by two terms: the loss $c_0(\mathbf{x}_0)$ of the original network, and the threshold ϵ related to $n_{l,AS}$. This loss function is the cross-entropy loss, not the classification error. However, it is believed that a small loss will result in a small classification error. Further, the result in Lemma 6.3.1 is valid for the fixed parameters in the pre-model. In practice, we can fine-tune the pre-model to achieve better accuracy.

Further, a small number of active neurons $n_{l,AS}$ is critical to get a high compress ratio. From Definition 6.3.1, $n_{l,AS}$ depends on the eigenvalue distribution of the covariance matrix $\mathbf{C}$. For a proper network structure and a good choice of the layer index l, if the eigenvalues of $\mathbf{C}$ are dominated by the first few eigenvalues, then $n_{l,AS}$ will be small. For instance, in Fig. 6.5(a), the eigenvalues for layers $4 \leq l \leq 7$ of VGG-19 are nearly exponential, decreasing to zero.

Algorithm 2: Training Procedure for the Active Subspace Network (ASNet)

Input: Pretrained deep neural network, layer index l, and number of active neurons r

Output: ASNet

1 **Initialize the active subspace layer.** The active subspace layer is a linear projection, where the projection matrix $\mathbf{V}_1 \in \mathbb{R}^{n \times r}$ is computed using Algorithm 3. If r is not specified, we use the default value $r = n_{\text{AS}}$ defined in Equation 6.12.

2 **Initialize the polynomial chaos expansion layer.** The polynomial chaos expansion layer is a nonlinear mapping from the reduced active subspace to the outputs, as shown in Equation 6.18. The weights $\mathbf{c}_{\boldsymbol{\alpha}}$ are computed using Equation 6.20.

3 **Construct the ASNet.** Combine the pre-model (the first l layers of the deep neural network) with the active subspace and polynomial chaos expansion layers to create a new network called ASNet.

4 **Fine-tuning.** Retrain all the parameters in the pre-model, active subspace layer, and polynomial chaos expansion layer in ASNet for several epochs using stochastic gradient descent.

6.3.3 Active Subspace Network (ASNet)

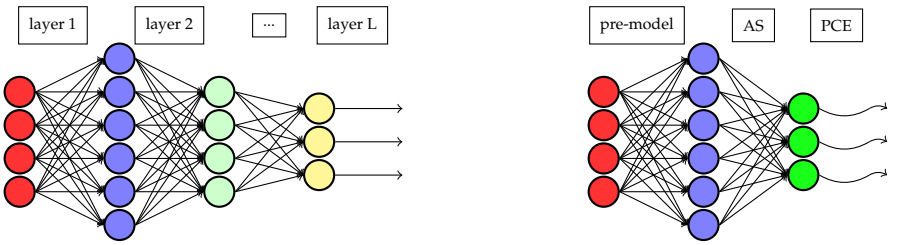
This subsection proposes a new network called ASNet that can reduce both the storage and computational cost. Given a deep neural network, we first choose a proper layer l and project the high-dimensional intermediate neurons to a low-dimensional vector in the active subspace. Afterward, the post-model is deleted completely and replaced with a nonlinear model that maps the low-dimensional active feature vector to the output directly. This new network, called ASNet, has three parts:

(1) **Pre-model:** the pre-model includes the first l layers of a deep neural network.

(2) **Active subspace layer:** a linear projection from the intermediate neurons to the low-dimensional active subspace.

(3) **Polynomial chaos expansion layer:** the polynomial chaos expansion [53, 175] maps the active-subspace variables to the output.

The initialization for the active subspace layer and polynomial chaos expansion layer are presented in Sections 6.3.4 and 6.3.5, respectively. We can also re-train all the parameters to increase the accuracy. The whole procedure is illustrated in Fig. 6.2 (b) and Algorithm 2.



(A) A deep neural network (B) The proposed ASNet

FIGURE 6.2: (a) The original deep neural network; (b) The proposed ASNet with three parts: a pre-model, an active subspace (AS) layer, and a polynomial chaos expansion (PCE) layer.

6.3.4 The Active Subspace Layer

This subsection presents an efficient method to project the high dimensional neurons to the active subspace. Given a dataset $\mathcal{D} = \{\mathbf{x}^1, \ldots, \mathbf{x}^m\}$, the empirical covariance matrix is computed by $\hat{\mathbf{C}} = \frac{1}{m}\sum_{i=1}^{m} \nabla c_l(\mathbf{x}^i)\nabla c_l(\mathbf{x}^i)^\top$. When ReLU is applied as an activation, $c_l(\mathbf{x})$ is not differentiable. In this case, ∇ denotes the sub-gradient with a little abuse of notation.

Instead of calculating the eigenvalue decomposition of $\hat{\mathbf{C}}$, we introduce a matrix $\hat{\mathbf{G}}$ and compute its singular value decomposition to save the computation cost:

$$\hat{\mathbf{G}} \triangleq [\nabla c_l(\mathbf{x}^1), \ldots, \nabla c_l(\mathbf{x}^m)] = \hat{\mathbf{V}}\hat{\mathbf{\Sigma}}\hat{\mathbf{U}}^\top \in \mathbb{R}^{n_l \times m} \text{ with } \hat{\mathbf{\Sigma}} = \text{diag}(\hat{\sigma}_1, \cdots, \hat{\sigma}_{n_l}). \tag{6.15}$$

The eigenvectors of $\mathbf{C}$ are approximated by the left singular vectors $\hat{\mathbf{V}}$ and the eigenvalues of $\mathbf{C}$ are approximated by the singular values of $\hat{\mathbf{G}}$, i.e., $\mathbf{\Lambda} \approx \hat{\mathbf{\Sigma}}^2$.

We use the memory-saving frequent direction method [54] to compute the r dominant singular value components, i.e., $\hat{\mathbf{G}} \approx \hat{\mathbf{V}}_r\hat{\mathbf{\Sigma}}_r\hat{\mathbf{U}}_r^\top$. Here r is smaller than the total number of samples. The frequent direction approach only stores an $n \times r$ matrix $\mathbf{S}$. In the beginning, each column of $\mathbf{S} \in \mathbb{R}^{n \times r}$ is initialized by a gradient vector. Then the randomized singular value decomposition [68] is used to generate $\mathbf{S} = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^\top$. Afterwards, $\mathbf{S}$ is updated in the following way,

$$\mathbf{S} \leftarrow \mathbf{V}\sqrt{\mathbf{\Sigma}^2 - \sigma_r^2}. \tag{6.16}$$

Now the last column of $\mathbf{S}$ is zero, and we replace it with the gradient vector of a new sample. By repeating this process, $\mathbf{S}\mathbf{S}^\top$ will approximate $\hat{\mathbf{G}}\hat{\mathbf{G}}^\top$ with high accuracy, and $\mathbf{V}$ will approximate the left singular vectors of $\hat{\mathbf{G}}$. The algorithm framework is presented in Algorithm 3.

After obtaining $\mathbf{\Sigma} = \text{diag}(\sigma_1, \ldots, \sigma_r)$, we can approximate the number of active neurons as

$$\hat{n}_{l,AS} = \arg\min \left\{ i : \frac{\sqrt{\sigma_1^2 + \ldots + \sigma_i^2}}{\sqrt{\sigma_1^2 + \ldots + \sigma_r^2}} \geq 1 - \epsilon \right\}. \tag{6.17}$$

Under the condition that $\sigma_i^2 \to \lambda_i$ for $i = 1, \ldots, r$ and $\lambda_i \to 0$ for $i = r +$

Algorithm 3: The frequent direction algorithm for computing the active subspace

Input: A dataset with m_{AS} input samples $\{\mathbf{x}_0^j\}_{j=1}^{m_{AS}}$, a pre-model $f_{\text{pre}}^l(\cdot)$, a subroutine for computing $\nabla c_l(\mathbf{x})$, and the dimension of truncated singular value decomposition r.

Output: The projection matrix $\mathbf{V} \in \mathbb{R}^{n_l \times r}$ and the singular values $\mathbf{\Sigma} \in \mathbb{R}^{r \times r}$.

Select r samples $\mathbf{x}_0^i$, compute $\mathbf{x}^i = f_{\text{pre}}^l(\mathbf{x}_0^i)$, and construct an initial matrix $\mathbf{S} \leftarrow [\nabla c_l(\mathbf{x}^1), \ldots, \nabla c_l(\mathbf{x}^r)]$.

while *the maximal number of samples m_{AS} is not reached* **do**

  Compute the singular value decomposition $\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T \leftarrow \text{svd}(\mathbf{S})$, where $\mathbf{\Sigma} = \text{diag}(\sigma_1, \ldots, \sigma_r)$.

  Update $\mathbf{S}$ by the soft-thresholding (6.16).

  Get a new sample $\mathbf{x}_0^{\text{new}}$, compute $\mathbf{x}^{\text{new}} = f_{\text{pre}}^l(\mathbf{x}_0^{\text{new}})$, and replace the last column of $\mathbf{S}$ (now all zeros) by the gradient vector $\mathbf{S}(:, r) \leftarrow \nabla c_l(\mathbf{x}^{\text{new}})$.

end

$1, \ldots, n_l$, (6.17) can approximate $n_{l,AS}$ in (6.12) with a high accuracy. Further, the projection matrix $\hat{\mathbf{V}}_1$ is chosen as the first $\hat{n}_{l,AS}$ columns of $\mathbf{V}$. The storage cost is reduced from $O(n_l^2)$ to $O(n_l r)$ and the computational cost is reduced from $O(n_l^2 r)$ to $O(n_l r^2)$.

6.3.5 Polynomial Chaos Expansion Layer

We continue to construct a new surrogate model to approximate the post-model of a deep neural network. This problem can be regarded as an uncertainty quantification problem if we set $\mathbf{z}$ as a random vector. We choose the nonlinear polynomial because it has higher expressive power than linear functions.

By the polynomial chaos expansion [174], the network output $\mathbf{y} \in \mathbb{R}^{n_L}$ is approximated by a linear combination of the orthogonal polynomial basis functions:

$$\hat{\mathbf{y}} \approx \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}} \boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z}), \text{ where } |\boldsymbol{\alpha}| = \alpha_1 + \ldots + \alpha_d. \tag{6.18}$$

Here $\boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z})$ is a multivariate polynomial basis function chosen based on the probability density function of $\mathbf{z}$. When the parameters $\mathbf{z} = [z_1, \ldots, z_r]^\top$ are independent, both the joint density function and the multi-variable basis function can be decomposed into products of one-dimensional functions, i.e., $\rho(\mathbf{z}) = \rho_1(z_1) \ldots \rho_r(z_r)$, $\boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z}) = \phi_{\alpha_1}(z_1)\phi_{\alpha_2}(z_2) \ldots \phi_{\alpha_r}(z_r)$. The marginal basis function $\phi_{\alpha_j}(z_j)$ is uniquely determined by the marginal density function $\rho_i(z_i)$. The scatter plot in Fig. 6.3 shows that the marginal probability density of ez_i is close to a Gaussian distribution.

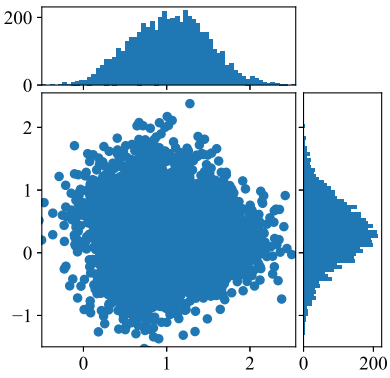


FIGURE 6.3: Distribution of the first two active subspace variables at the 6-th layer of VGG-19 for CIFAR-10.

Suppose $\rho_i(z_i)$ follows a Gaussian distribution, then $\phi_{\alpha_j}(z_j)$ will be a Hermite polynomial [104], i.e.,

$$\phi_0(z) = 1,\ \phi_1(z) = z,\ \phi_2(z) = 4z^2 - 2,\ \phi_{p+1}(z) = 2z\phi_p(z) - 2p\phi_{p-1}(z). \tag{6.19}$$

In general, the elements in $\mathbf{z}$ can be non-Gaussian correlated. In this case, the basis functions $\{\boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z})\}$ can be built via the Gram-Schmidt approach described in [34].

The coefficient $\mathbf{c}_{\boldsymbol{\alpha}}$ can be computed by a linear least-square optimization. Denote $\mathbf{z}^j = \hat{\mathbf{V}}_1^\top f_{\text{pre}}^l(\mathbf{x}_0^j)$ as the random samples and $\mathbf{y}^j$ as the network output for $j = 1, \ldots, m_{\text{PCE}}$. The coefficient vector $\mathbf{c}_{\boldsymbol{\alpha}}$ can be computed by

$$\min_{\{\mathbf{c}_{\boldsymbol{\alpha}}\}} \quad \frac{1}{m_{\text{PCE}}} \sum_{j=1}^{m_{\text{PCE}}} \|\mathbf{y}^j - \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}} \boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z}^j)\|^2. \tag{6.20}$$

Based on the Nyquist-Shannon sampling theorem, the number of samples to train $\mathbf{c}_{\boldsymbol{\alpha}}$ needs to satisfy $m_{\text{PCE}} \geq 2n_{\text{basis}} = 2\binom{r+p}{p}$. However, this number can be reduced to a smaller set of "important" samples by the D-optimal design [181] or the sparse regularization approach [33].

The polynomial chaos expansion builds a surrogate model to approximate the deep neural network output $\mathbf{y}$. This idea is similar to the knowledge distillation [80], where a pre-trained teacher network teaches a smaller student network to learn the output feature. However, our polynomial-chaos layer uses one nonlinear projection, whereas the knowledge distillation uses a series of layers. Therefore, the polynomial chaos expansion is more efficient in terms of computational and storage costs. The polynomial chaos expansion layer is different from the polynomial activation because the dimension of $\mathbf{z}$ may be different from that of output $\mathbf{y}$.

The problem (6.20) is convex, and any first-order method can get a globally optimal solution. Denote the optimal coefficients as $\mathbf{c}_{\alpha}^*$ and the finial objective

value as δ^*, i.e.,

$$\delta^* = \frac{1}{m_{\text{PCE}}} \sum_{j=1}^{m_{\text{PCE}}} \|\mathbf{y}^j - \psi^*(\mathbf{z}^j)\|^2, \text{ where } \psi^*(\mathbf{z}^j) = \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}}^* \boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z}^j). \tag{6.21}$$

If $\delta^* = 0$, the polynomial chaos expansion is a good approximation to the original deep neural network on the training dataset. However, the approximation loss of the testing dataset may be large because of the overfitting phenomena.

The objective function in (6.20) is an empirical approximation to the expected error

$$\mathbb{E}_{(\mathbf{z},\mathbf{y})}[\|\mathbf{y} - \psi(\mathbf{z})\|^2], \text{ where } \psi(\mathbf{z}) = \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}} \boldsymbol{\phi}_{\boldsymbol{\alpha}}(\mathbf{z}). \tag{6.22}$$

According to Hoeffding's inequality [82], the expected error (6.22) is close to the empirical error (6.20) with a high probability. Consequently, the loss for ASNet with a polynomial chaos expansion layer is bounded as follows.

Lemma 6.3.2 *Suppose that the optimal solution for the problem (6.20) is $\mathbf{c}_\alpha^*$, the optimal polynomial chaos expansion is $\psi^*(\mathbf{z})$, and the optimal residue is δ^*. Assume that there exist consts a, b such that for all j, $\|\mathbf{y}^j - \psi^*(\mathbf{z}^j)\|^2 \in [a, b]$. Then the loss of ASNet will be upper bounded*

$$\mathbb{E}_{\mathbf{z}}[(loss(\psi^*(\mathbf{z})))^2] \leq 2\mathbb{E}_{\mathbf{x}_0}[(c_0(\mathbf{x}_0))^2] + 2n_L(\delta^* + t) \ \ w.p.\ 1 - \gamma^*, \tag{6.23}$$

where t is a user-defined threshold, and $\gamma^* = \exp(-\frac{2t^2 m_{PCE}}{(b-a)^2})$.

Proof Since the cross entropy loss function is $\sqrt{n_L}$-Lipschitz continuous, we have

$$\mathbb{E}_{(\mathbf{y},\mathbf{z})}[(\text{loss}(\mathbf{y}) - \text{loss}(\psi^*(\mathbf{z})))^2] \leq n_L \mathbb{E}_{(\mathbf{y},\mathbf{z})}[\|\mathbf{y} - \psi^*(\mathbf{z})\|^2], \tag{6.24}$$

Denote $\mathcal{T}^j = \|\mathbf{y}^j - \psi^*(\mathbf{z}^j)\|^2$ for $i = 1, \ldots, n_L$. $\{\mathcal{T}^j\}$ are independent under the assumption that the data samples are independent. By Hoeffding's inequality, for any constant t, it holds that

$$\mathbb{E}[\mathcal{T}] \leq \frac{1}{m_{\text{PCE}}} \sum_j \mathcal{T}^j + t \quad \text{w.p. } 1 - \gamma^*, \tag{6.25}$$

with $\gamma^* = \exp(-\frac{2t^2 m_{\text{PCE}}}{(b-a)^2})$. Equivalently,

$$\mathbb{E}_{(\mathbf{y},\mathbf{z})}[\|\mathbf{y} - \psi^*(\mathbf{z})\|^2] \leq \delta^* + t \quad \text{w.p. } 1 - \gamma^*, \tag{6.26}$$

Consequently, there is

$$\begin{aligned}&\mathbb{E}_{\mathbf{z}}[(\text{loss}(\psi^*(\mathbf{z})))^2]\\ \leq& 2\mathbb{E}_{\mathbf{y}}[(\text{loss}(\mathbf{y}))^2] + 2\mathbb{E}_{(\mathbf{y},\mathbf{z})}[(\text{loss}(\psi^*(\mathbf{z})) - \text{loss}(\mathbf{y}))^2]\\ \leq& 2\mathbb{E}_{\mathbf{x}_0}[(c_0(\mathbf{x}_0))^2] + 2n_L(\delta^* + t) \ \text{ w.p. } 1-\gamma^*.\end{aligned}$$

The last inequality follows from $c_0(\mathbf{x}_0) = c_l(\mathbf{x}_l) = \text{loss}(\mathbf{y})$, equations (6.24) and (6.26). This completes the proof.

Lemma 6.3.2 shows with a high probability $1-\gamma^*$, the expected error of ASNet without fine-tuning is bounded by the pre-trained error of the original network, the accuracy loss in solving the polynomial chaos subproblem (6.21), and the number of classes n_L. The probability γ^* is controlled by the threshold t as well as the number of training samples m_{PCE}.

In practice, we always re-train ASNet for several epochs and the accuracy of ASNet is beyond the scope of Lemma 6.3.2.

6.3.6 Structured Re-training of ASNet

The pre-model can be further compressed by various techniques such as network pruning and sharing [70], low-rank factorization [124, 101, 51], or data quantization [38, 31]. Denote $\boldsymbol{\theta}$ as the weights in ASNet and $\{\mathbf{x}_0^1, \ldots, \mathbf{x}_0^m\}$ as the training dataset. Here, $\boldsymbol{\theta}$ denotes all the parameters in the pre-model, active subspace layer, and the polynomial chaos expansion layer. We re-train the network by solving the following regularized optimization problem:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{m}\sum_{i=1}^{m} \text{loss}(f(\boldsymbol{\theta}; \mathbf{x}_0^i)) + \lambda R(\boldsymbol{\theta}). \tag{6.27}$$

Here $(\mathbf{x}_0^i, \mathbf{y}^i)$ is a training sample, m is the total number of training samples, $\text{loss}(\cdot)$ is the cross-entropy loss function, $R(\boldsymbol{\theta})$ is a regularization function, and λ is a regularization parameter. Different regularization functions can result in different model structures. For instance, an ℓ_1 regularizer $R(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$ [2, 150, 176] will return a sparse weight, an $\ell_{1,2}$-norm regularizer will result in column-wise sparse weights, a nuclear norm regularizer will result in low-rank weights. At each iteration, we solve (6.27) by a stochastic proximal gradient decent algorithm [155]

$$\boldsymbol{\theta}^{k+1} = \operatorname*{argmax}_{\boldsymbol{\theta}} \quad (\boldsymbol{\theta} - \boldsymbol{\theta}^k)^\top \mathbf{g}^k + \frac{1}{2\alpha_k}\|\boldsymbol{\theta} - \boldsymbol{\theta}^k\|_2^2 + \lambda R(\boldsymbol{\theta}). \tag{6.28}$$

Here $\mathbf{g}^k = \frac{1}{|\mathcal{B}_k|}\sum_{i\in\mathcal{B}_k} \nabla_{\boldsymbol{\theta}} \text{loss}(f(\boldsymbol{\theta}; \mathbf{x}_0^i), \mathbf{y}^i)$ is the stochastic gradient, $\mathcal{B}_k$ is a batch at the k-th step, and α_k is the stepsize.

In this work, we chose the ℓ_1 regularization to get sparse weight matrices. In this case, problem (6.28) has a closed-form solution:

$$\boldsymbol{\theta}^{k+1} = \mathcal{S}_{\alpha_k\lambda}(\boldsymbol{\theta}^k - \alpha_k \mathbf{g}^k), \tag{6.29}$$

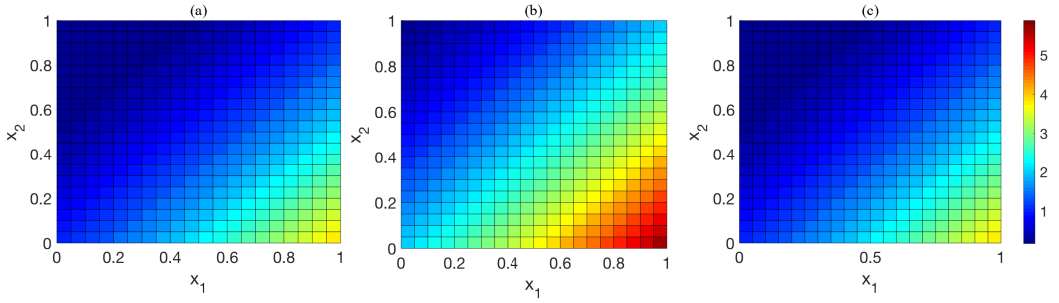


FIGURE 6.4: Perturbations along the directions of an active-subspace direction and of principal component, respectively. (a) The function $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} - b$. (b) The perturbed function along the active-subspace direction. (c) The perturbed function along the principal component analysis direction.

where $\mathcal{S}_\lambda(\mathbf{x}) = \mathbf{x} \odot \max(0, 1 - \lambda/|\mathbf{x}|)$ is a soft-thresholding operator.

6.4 Active-Subspace for Universal Adversarial Attacks

This section investigates how to generate a universal adversarial attack by the active-subspace method. Given a function $f(\mathbf{x})$, the maximal perturbation direction is defined by

$$\mathbf{v}_\delta^* = \underset{\|\mathbf{v}\|_2 \le \delta}{\operatorname{argmax}} \quad \mathbb{E}_{\mathbf{x}}[(f(\mathbf{x}+\mathbf{v}) - f(\mathbf{x}))^2]. \tag{6.30}$$

Here, δ is a user-defined perturbation upper bound. By the first order Taylor expansion, we have $f(\mathbf{x}+\mathbf{v}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T\mathbf{v}$, and problem (6.30) can be reduced to

$$\mathbf{v}_{AS} = \underset{\|\mathbf{v}\|_2=1}{\operatorname{argmax}} \quad \mathbb{E}_{\mathbf{x}}[(\nabla f(\mathbf{x})^T\mathbf{v})^2] = \underset{\|\mathbf{v}\|_2=1}{\operatorname{argmax}} \quad \mathbf{v}^T\mathbb{E}_{\mathbf{x}}[\nabla f(\mathbf{x})\nabla f(\mathbf{x})^T]\mathbf{v}. \tag{6.31}$$

The vector $\mathbf{v}_{AS}$ is exactly the dominant eigenvector of the covariance matrix of $\nabla f(\mathbf{x})$. The solution for (6.30) can be approximated by $+\delta\mathbf{v}_{AS}$ or $-\delta\mathbf{v}_{AS}$. Here, both $\mathbf{v}_{AS}$ and $-\mathbf{v}_{AS}$ are solutions of (6.31) but their effect on (6.30) are different.

Example 6.4.1 *Consider a two-dimensional function $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} - b$ with $\mathbf{a} = [1, -1]^T$ and $b = 1$, and $\mathbf{x}$ follows a uniform distribution in a two-dimensional square domain $[0, 1]^2$, as shown in Fig. 6.4 (a). It follows from direct computations that $\nabla f(\mathbf{x}) = \mathbf{a}$ and the covariance matrix $\mathbf{C} = \mathbf{a}\mathbf{a}^T$. The dominant eigenvector of $\mathbf{C}$ or the active-subspace direction is $\mathbf{v}_{AS} = \mathbf{a}/\|\mathbf{a}\|_2 = [1/\sqrt{2}, -1/\sqrt{2}]$. We apply $\mathbf{v}_{AS}$ to perturb $f(\mathbf{x})$ and plot $f(\mathbf{x} + \delta\mathbf{v}_{AS})$ in Fig. 6.4 (b), which shows a significant difference even for a small permutation $\delta = 0.3$. Furthermore, we plot the perturbed function along the first principal component direction $\mathbf{w}_1 = [1/\sqrt{2}, 1/\sqrt{2}]^T$ in Fig. 6.4 (c).*

Here, $\mathbf{w}_1$ *is the eigenvector of the covariance matrix* $\mathbb{E}_{\mathbf{x}}[\mathbf{x}\mathbf{x}^T] = \begin{bmatrix} 1/3 & 1/4 \\ 1/4 & 1/3 \end{bmatrix}$. *However,* $\mathbf{w}_1$ *does not result in any perturbation because* $\mathbf{a}^T\mathbf{w}_1 = 0$. *This example indicates the difference between the active-subspace and principal component analysis: the active-subspace direction can capture the sensitivity information of* $f(\mathbf{x})$, *whereas the principal component is independent of* $f(\mathbf{x})$.

6.4.1 Universal Perturbation of Deep Neural Networks

Given a dataset $\mathcal{D}$ and a classification function $j(\mathbf{x})$ that maps an input sample to an output label. The universal perturbation seeks a vector $\mathbf{v}^*$ whose norm is upper bounded by δ, such that the class label can be perturbed with a high probability, i.e.,

$$\mathbf{v}^* = \underset{\|\mathbf{v}\| \leq \delta}{\operatorname{argmax}} \operatorname{prob}_{\mathbf{x} \in \mathcal{D}}[j(\mathbf{x}+\mathbf{v}) \neq j(\mathbf{x})] = \underset{\|\mathbf{v}\| \leq \delta}{\operatorname{argmax}} \mathbb{E}_{\mathbf{x}}[1_{j(\mathbf{x}+\mathbf{v}) \neq j(\mathbf{x})}], \quad (6.32)$$

where 1_d equals one if the condition d is satisfied and zero otherwise. Solving problem (6.32) directly is challenging because both 1_d and $j(\mathbf{x})$ are discontinuous. By replacing $j(\mathbf{x})$ with the loss function $c(\mathbf{x}) = \operatorname{loss}(f(\mathbf{x}))$ and the indicator function 1_d with a quadratic function, we reformulate problem (6.32) as

$$\max_{\mathbf{v}} \quad \mathbb{E}_{\mathbf{x}}[(c(\mathbf{x}+\mathbf{v}) - c(\mathbf{x}))^2] \quad \text{s.t.} \quad \|\mathbf{v}\|_2 \leq \delta. \quad (6.33)$$

The ball-constrained optimization problem (6.33) can be solved by various numerical techniques such as the spectral gradient descent method [12], and the limited-memory projected quasi-Newton [151]. However, these methods can only guarantee convergence to a local stationary point. Instead, we are interested in computing a direction that can achieve a better objective value by a heuristic algorithm.

6.4.2 Recursive Projection Method

Using the first order Taylor expansion $c(\mathbf{x}+\mathbf{v}) \approx c(\mathbf{x}) + \mathbf{v}^T\nabla c(\mathbf{x})$, we reformulate problem (6.33) as a ball constrained quadratic problem

$$\max_{\mathbf{v}} \quad \mathbf{v}^T\mathbb{E}_{\mathbf{x}}[\nabla c(\mathbf{x})\nabla c(\mathbf{x})^T]\mathbf{v} \quad \text{s.t.} \quad \|\mathbf{v}\|_2 \leq \delta. \quad (6.34)$$

Problem (6.34) is easy to solve because its closed-form solution is exactly the dominant eigenvector of the covariance matrix $\mathbf{C} = \mathbb{E}_{\mathbf{x}}[\nabla c(\mathbf{x})\nabla c(\mathbf{x})^T]$ or the first active-subspace direction. However, the dominant eigenvector in (6.34) may not be efficient because $c(\mathbf{x})$ is nonlinear. Therefore, we compute $\mathbf{v}$ recursively by

$$\mathbf{v}^{k+1} = \operatorname{proj}(\mathbf{v}^k + s^k d_{\mathbf{v}}^k), \quad (6.35)$$

where $\text{proj}(\mathbf{v}) = \mathbf{v} \times \min(1, \delta/\|\mathbf{v}\|_2)$[1], s^k is the stepsize, and $d_{\mathbf{v}}^k$ is approximated by

$$d_{\mathbf{v}}^k = \underset{d_{\mathbf{v}}}{\text{argmax}} \quad d_{\mathbf{v}}^T \mathbb{E}_{\mathbf{x}} \left[\nabla c \left(\mathbf{x} + \mathbf{v}^k \right) \nabla c \left(\mathbf{x} + \mathbf{v}^k \right)^T \right] d_{\mathbf{v}}, \text{ s.t. } \|d_{\mathbf{v}}\|_2 \leq 1. \quad (6.36)$$

Namely, $d_{\mathbf{v}}^k$ is the dominant eigenvector of $\mathbf{C}^k = \mathbb{E}_{\mathbf{x}} \left[\nabla c \left(\mathbf{x} + \mathbf{v}^k \right) \nabla c \left(\mathbf{x} + \mathbf{v}^k \right)^T \right]$. Because $d_{\mathbf{v}}^k$ maximizes the changes in $\mathbb{E}_{\mathbf{x}}[(c(\mathbf{x} + \mathbf{v} + d_{\mathbf{v}}) - c(\mathbf{x} + \mathbf{v}))^2]$, we expect that the attack ratio keeps increasing, i.e., $r(\mathbf{v}^{k+1}; \mathcal{D}) \geq r(\mathbf{v}^k; \mathcal{D})$, where

$$r(\mathbf{v}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x}^i \in \mathcal{D}} 1_{j(\mathbf{x}^i + \mathbf{v}) \neq j(\mathbf{x}^i)}. \quad (6.37)$$

The backtracking line search approach [6] is employed to choose s^k such that the attack ratio of $\mathbf{v}^k + s^k d_{\mathbf{v}}^k$ is higher than the attack ratio of both $\mathbf{v}^k$ and $\mathbf{v}^k - s^k d_{\mathbf{v}}^k$, i.e.,

$$s^k = \min_i \{ s_{i,t}^k : r(\mathbf{v}_{i,t}^{k+1}; \mathcal{D}) > \max(r(\mathbf{v}_{i,-t}^{k+1}; \mathcal{D}), r(\mathbf{v}^k; \mathcal{D})\}, \quad (6.38)$$

where $s_{i,t}^k = (-1)^t s_0 \gamma^i$, $t \in \{1, -1\}$, s_0 is the initial stepsize, $\gamma < 1$ is the decrease ratio, and $\mathbf{v}_{i,t}^{k+1} = \text{proj}(\mathbf{v}^k + s_{i,t}^{k+1} d_{\mathbf{v}}^k)$. If such a stepsize s^k exists, we update $\mathbf{v}^{k+1}$ by (6.35) and repeat the process. Otherwise, we record the number of failures and stop the algorithm when the number of failures is greater than the threshold.

The overall flow is summarized in Algorithm 4. In practice, instead of using the whole dataset to train this attack vector, we use a subset $\mathcal{D}^0$. The impact for a different number of samples is discussed in section 6.5.2.2.

6.5 Numerical Experiments

In this section, we show the power of active subspace in revealing the number of active neurons, compressing neural networks, and computing the universal adversarial perturbation. All codes are implemented in PyTorch and are available online[2].

6.5.1 Structural Analysis and Compression

We test the ASNet constructed by Algorithm 2, and set the polynomial order as $p = 2$, the number of active neurons as $r = 50$, and the threshold in Equation equation 6.12 as $\epsilon = 0.05$ on default. Inspired by the knowledge distillation [80], we re-train all the parameters in the ASNet by minimizing

[1] A better option is to project v on a sphere, but experiments are computed for the projection into a ball.

[2] https://github.com/chunfengc/ASNet

Algorithm 4: Recursive Active Subspace Universal Attack

Input: A pre-trained deep neural network denoted as $c(\mathbf{x})$, a classification oracle $j(\mathbf{x})$, a training dataset $\mathcal{D}^0$, an upper bound for the attack vector δ, an initial stepsize s_0, a decrease ratio $\gamma < 1$, and the parameter in the stopping criterion α.

Output: The universal active adversarial attack vector $\mathbf{v}_{AS}$.

Initialize the attack vector as $\mathbf{v}^0 = 0$.
for $k = 0, 1, \ldots$ **do**
  Select the training dataset as $\mathcal{D} = \{\mathbf{x}^i + \mathbf{v}^k : \mathbf{x}^i \in \mathcal{D}^0 \text{ and } j(\mathbf{x}^i + \mathbf{v}^k) = j(\mathbf{x}^i)\}$, then compute the dominate active subspace direction $d_{\mathbf{v}}$ using Algorithm 3.
  for $i = 0, 1, \ldots, l$ **do**
    Let $s^k_{i,\pm} = (-1)^{\pm} s_0 \gamma^i$ and $\mathbf{v}^{k+1}_{i,\pm} = \text{proj}(\mathbf{v}^k + s^{k+1}_{i,\pm} d^k_{\mathbf{v}})$. Compute the attack ratios $r(\mathbf{v}^{k+1}_{i,1})$ and $r(\mathbf{v}^{k+1}_{i,-1})$ by (6.37).
    If either $r(\mathbf{v}^{k+1}_{i,1})$ or $r(\mathbf{v}^{k+1}_{i,-1})$ is greater than $r(\mathbf{v}^k)$, stop the process.
    Return $s^k = (-1)^t s^k_{i,1}$, where $t = 1$ if $r(\mathbf{v}^{k+1}_{i,1}) \geq r(\mathbf{v}^{k+1}_{i,-1})$ and $t = -1$ otherwise.
  end
  If no stepsize s^k is returned, let $s^k = s_0 r^I$ and record this step as a failure.
  Compute the next iteration $\mathbf{v}^{k+1}$ by the projection (6.35).
  If the number of failure is greater the threshold α, stop.
end

the following loss function

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{m} \beta H\left(\text{ASNet}_{\boldsymbol{\theta}}(\mathbf{x}^i_0), f(\mathbf{x}^i_0)\right) + (1-\beta) H\left(\text{ASNet}_{\boldsymbol{\theta}}(\mathbf{x}^i_0), \mathbf{y}^i\right).$$

Here, the cross entropy $H(\mathbf{p}, \mathbf{q}) = \sum_j s(\mathbf{p})_j \log s(\mathbf{q})_j$, the softmax function $s(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$, and the parameter $\beta = 0.1$ on default. We re-train ASNet for 50 epochs by ADAM [96]. The stepsizes for the pre-model are set as 10^{-4} and 10^{-3} for VGG-19 and ResNet, and the stepsize for the active subspace layer and the polynomial chaos expansion layer is set as 10^{-5}, respectively,

We also seek sparser weights in ASNet by the proximal stochastic gradient descent method in Section 6.3.6. On default, we set the stepsize as 10^{-4} for the pre-model and 10^{-5} for the active subspace layer and the polynomial chaos expansion layer. The maximal epoch is set as 100. The obtained sparse model is denoted as ASNet-s.

In all figures and tables, the numbers in the bracket of ASNet(·) or ASNet-s(·) indicate the index of a cut-off layer. We report the performance for different cut-off layers in terms of *accuracy, storage, and computational complexities*.

TABLE 6.1: Comparison of number of neurons r of VGG-19 on CIFAR-10. For the storage speedup, the higher is better. For the accuracy reduction before or after fine-tuning, the lower is better.

	$r = 25$				$r = 50$				$r = 75$			
	ϵ	Storage	Acc. Reduce		ϵ	Storage	Acc. Reduce		ϵ	Storage	Acc. Reduce	
			Before	After			Before	After			Before	After
ASNet(5)	0.34	**20.7**×	7.06	2.82	0.18	14.4×	4.40	1.82	0.11	11.0×	**3.64**	**1.66**
ASNet(6)	0.24	**12.8**×	2.14	0.59	0.11	10.1×	1.62	0.27	0.05	8.3×	**1.40**	**0.21**
ASNet(7)	0.15	**9.3**×	0.79	0.11	0.06	7.8×	**0.63**	**-0.10**	0.03	6.7×	0.77	0.00

6.5.1.1 Choices of Parameters

We first show the influence of the number of reduced neurons r, tolerance ϵ, and cutting-off layer index l of VGG-19 on CIFAR-10 in Table 6.1. The VGG-19 can achieve 93.28% testing accuracy with 76.45 Mb storage consumption. Here, $\epsilon = \frac{\lambda_{r+1}+\ldots+\lambda_n}{\lambda_1+\ldots+\lambda_n}$. For different choices of r, we display the corresponding tolerance ϵ, the storage speedup compared with the original teacher network, and the testing accuracy reduction for ASNet before and after fine-tuning compared with the original teacher network.

Table 6.1 shows that when the cutting-off layer is fixed, a larger r usually results in a smaller tolerance ϵ and a smaller accuracy reduction but also a smaller storage speedup. This corresponds to Lemma 6.3.1, that the error of ASNet before fine-tuning is upper bounded by $O(\epsilon)$. Comparing $r = 50$ with $r = 75$, we find that $r = 50$ can achieve almost the same accuracy with $r = 75$ with a higher storage speedup. $r = 50$ can even achieve better accuracy than $r = 75$ in layer 7 probably because of overfitting. This guides us to choose $r = 50$ in the following numerical experiments. For different layers, we see a later cutting-off layer index can produce a lower accuracy reduction but a smaller storage speedup. In other words, the choice of layer index is a trade-off between accuracy reduction with storage speedup.

6.5.1.2 Efficiency of the ASNet

We show the effectiveness of ASNet constructed by Steps 1-3 of Algorithm 2 without fine-tuning. We investigate the following three properties. (1) **Redundancy of neurons.** The distributions of the first 200 singular values of the matrix $\hat{\mathbf{G}}$ (defined in equation 6.15) are plotted in Fig. 6.5 (a). The singular values decrease almost exponentially for layers $l \in \{4, 5, 6, 7\}$. Although the total numbers of neurons are 8192, 16384, 16384, and 16384, the numbers of active neurons are only 105, 84, 54, and 36, respectively. (2) **Redundancy of the layers.** We cut off the deep neural network at an intermediate layer and replace the subsequent layers with one simple logistic regression [84]. As shown by the red bar in Fig. 6.5 (b), the logistic regression can achieve relatively high accuracy. This verifies that the features trained from the first few layers already have a high expression power since replacing all subsequent layers with a simple expression loses little accuracy. (3) **Efficiency of the active-subspace and polynomial chaos expansion.** We compare the proposed

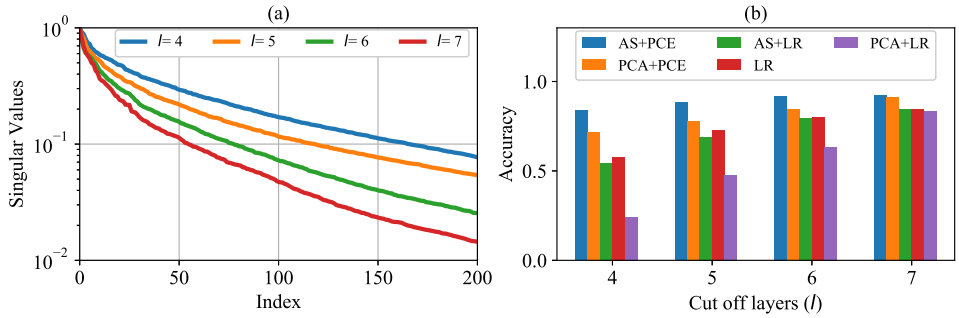


FIGURE 6.5: Structural analysis of VGG-19 on the CIFAR-10 dataset. (a) The first 200 singular values for layers $4 \leq l \leq 7$; (b) The accuracy (without any fine-tuning) obtained by active-subspace (AS) and polynomial chaos expansions (PCE) compared with principal component analysis (PCA) and logistic regression (LR).

active-subspace layer with the principal component analysis [90] in projecting the high-dimensional neuron to a low-dimensional space and also compare the polynomial chaos expansion layer with logistic regression in terms of their efficiency to extract class labels from the low-dimensional variables. Fig. 6.5 (b) shows that the combination of active-subspace and polynomial chaos expansion can achieve the best accuracy.

6.5.1.3 CIFAR-10

We continue to present the results of ASNet and ASNet-s on CIFAR-10 by two widely used networks: VGG-19 and ResNet-110, in Tables 6.2 and 6.3, respectively. The second column shows the testing accuracy for the corresponding network. We report the storage and computational costs for the pre-model, and post-model (i.e., active-subspace plus polynomial chaos expansion for ASNet and ASNet-s), and overall results, respectively. For both examples, ASNet and ASNet-s can achieve a similar accuracy with the teacher network yet with much smaller storage and computational cost. For VGG-19, ASNet achieves 14.43× storage savings and 3.44× computational reduction; ASNet-s achieves 23.98× storage savings and 7.30× computational reduction. For most ASNet and ASNet-s networks, the storage and computational costs of the post-models achieve significant performance boosts by our proposed network structure changes. It is not surprising to see that increasing the layer index (i.e., cutting off the deep neural network at a later layer) can produce a higher accuracy. However, increasing the layer index also results in a smaller compression ratio. In other words, the choice of layer index is a trade-off between the accuracy reduction and the compression ratio.

For Resnet-110, our results are not as good as those on VGG-19. We find that the eigenvalues for its covariance matrix are not exponentially decreasing as that of VGG-19, which results in a large number of active neurons or a large error ϵ when fixing $r = 50$. A possible reason is that ResNet updates as

TABLE 6.2: Accuracy and storage on VGG-19 for CIFAR-10. Here, "Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, "AS" and "PCE" denote the active subspace and polynomial chaos expansion layer, respectively.

Network	**Accuracy**	**Storage (MB)**			**Flops (10^6)**		
VGG-19	93.28%		76.45			398.14	
		Pre-M	**AS+PCE**	**Overall**	**Pre-M**	**AS+PCE**	**Overall**
ASNet(5)	91.46%	2.12	3.18	5.30	115.02	0.83	115.85
			(23.41×)	(14.43×)		(340.11×)	(3.44×)
ASNet-s(5)	90.40%	1.14	2.05	**3**.19	54.03	0.54	**5**4.56
		(1.86×)	(36.33×)	**(**23.98×)	(2.13×)	(527.91×)	**(**7.30×)
ASNet(6)	93.01%	4.38	3.18	7.55	152.76	0.83	153.60
			(22.70×)	(10.12×)		(294.76×)	(2.59×)
ASNet-s(6)	91.08%	1.96	1.81	3.77	67.37	0.48	67.85
		(2.24×)	(39.73×)	(20.27×)	(2.27×)	(515.98×)	(5.87×)
ASNet(7)	**9**3.38%	6.63	3.18	9.80	190.51	0.83	191.35
			(21.99×)	(7.80×)		(249.41×)	(2.08×)
ASNet-s(7)	90.87%	2.61	1.91	4.52	80.23	0.50	80.73
		(2.54×)	(36.64×)	(16.92×)	(2.37×)	(415.68×)	(4.93×)

$\mathbf{x}_{l+1} = \mathbf{x}_l + f_l(\mathbf{x}_l)$. Hence, the partial gradient $\partial \mathbf{x}_{l+1}/\partial \mathbf{x}_l = I + \nabla f_l(\mathbf{x}_l)$ is less likely to be low-rank.

6.5.1.4 CIFAR-100

Next, we present the results of VGG-19 and ResNet-110 on CIFAR-100 in Tables 6.4 and 6.5, respectively. On VGG-19, ASNet can achieve 7.45× storage savings and 2.08× computational reduction, and ASNet-s can achieve 9.06× storage savings and 2.73× computational reduction. The accuracy loss is negligible for VGG-19 but larger for ResNet-110. The performance boost of ASNet is obtained by just changing the network structures and without any model compression (e.g., pruning, quantization, or low-rank factorization).

6.5.2 Universal Adversarial Attacks

This subsection demonstrates the effectiveness of active subspace in identifying a universal adversarial attack vector. We denote the result generated by Algorithm 4 as "AS" and compare it with the "UAP" method in [120] and with "random" Gaussian distribution vector. The parameters in Algorithm 4 are set as $\alpha = 10$ and $\delta = 5, \ldots, 10$. The default parameters of UAP are applied except for the maximal iteration. In the implementation of [120], the maximal iteration is set as infinity, which is time-consuming when the training dataset or the number of classes is large. In our experiments, we set the maximal iteration as 10. In all figures and tables, we report the average attack ratio and CPU time in training out of ten repeated experiments with different training datasets. A higher attack ratio means the corresponding algorithm is better at fooling the given deep neural network. The datasets are chosen in two ways. We first test data points from one class (e.g., trousers in Fashion-MNIST) because these data points share lots of common features and have a higher

TABLE 6.3: Accuracy and storage on ResNet-110 for CIFAR-10. Here, "Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, "AS" and "PCE" denote the active subspace and polynomial chaos expansion layer, respectively.

Network	**Accuracy**	**Storage (MB)**			**Flops (10^6)**		
ResNet-110	93.78%		6.59			252.89	
		Pre-M	**AS+PCE**	**Overall**	**Pre-M**	**AS+PCE**	**Overall**
ASNet(61)	89.56%	1.15	1.61 (3.37×)	2.77 (2.38×)	140.82	0.42 (265.03×)	141.24 (1.79×)
ASNet-s(61)	89.26%	0.83 (1.39×)	1.23 (4.41×)	**2**.06 (3.19×)	104.05 (1.35×)	0.32 (346.82×)	**1**04.37 (**2**.42×)
ASNet(67)	90.16%	1.37	1.61 (3.24×)	2.98 (2.21×)	154.98	0.42 (231.55×)	155.40 (1.63×)
ASNet-s(67)	89.69%	1.00 (1.36×)	1.22 (4.29×)	2.22 (2.97×)	116.38 (1.33×)	0.32 (306.72×)	116.70 (2.17×)
ASNet(73)	**9**0.48%	1.58	1.61 (3.11×)	3.19 (2.06×)	169.13	0.42 (198.07×)	169.55 (1.49×)
ASNet-s(73)	90.02%	1.18 (1.34×)	1.16 (4.32×)	2.34 (2.82×)	128.65 (1.31×)	0.30 (275.74×)	128.96 (1.96×)

probability of being attacked by a universal perturbation vector. We then conduct experiments on the whole dataset to show our proposed algorithm can also provide better performance compared with the baseline even if the dataset has diverse features.

6.5.2.1 Fashion-MNIST

Firstly, we present the adversarial attack result on Fashion-MNIST by a 4-layer neural network. There are two convolutional layers with kernel size equal 5×5. The size of output channels for each convolutional layer is 20 and 50, respectively. Each convolutional layer is followed by a ReLU activation layer and a max-pooling layer with a kernel size of 2×2. There are two fully connected layers. The first fully connected layer has the dimensionality of input and output features equal to 800 and 500, respectively.

Fig. 6.6 presents the attack ratio of our active-subspace method compared with the baselines UAP method [120] and Gaussian random vectors. The top figures show the results for just one class (i.e., trousers), and the bottom figures show the results for all ten classes. For all perturbation norms, the active-subspace method can achieve around 30% higher attack ratio than UAP while more than 10 times faster. This verifies that the active-subspace method has better universal representation ability compared with UAP because the active-subspace can find a universal direction while UAP solves data-dependent subproblems independently. By the active-subspace approach, the attack ratio for the first class and the whole dataset are around 100% and 75%, respectively. This coincides with our intuition that the data points in one class have higher similarity than data points from different classes.

In Fig. 6.7, we plot one image from Fashion-MNIST and its perturbation by the active-subspace attack vector. The attacked image in Fig. 6.7 (c) still looks

TABLE 6.4: Accuracy and storage on VGG-19 for CIFAR-100. Here, "Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, "AS" and "PCE" denote the active subspace and polynomial chaos expansion layer, respectively.

Network	**Top-1**	**Top-5**	**Storage (MB)**			**Flops (10^6)**		
VGG-19	71.90%	89.57%	76.62			398.18		
			Pre-M	**AS+PCE**	**Overall**	**Pre-M**	**AS+PCE**	**Overall**
ASNet(7)	70.77%	91.05%	6.63	3.63 (19.23×)	10.26 (7.45×)	190.51	0.83 (249.41×)	191.35 (2.08×)
ASNet-s(7)	70.20%	90.90%	5.20 (1.27×)	3.24 (21.56×)	8.44 (9.06×)	144.81 (1.32×)	0.85 (244.57×)	**1**45.66 (**2**.73×)
ASNet(8)	69.50%	90.15%	8.88	1.29 (52.50×)	10.17 (7.52×)	228.26	0.22 (779.04×)	228.48 (1.74×)
ASNet-s(8)	69.17%	89.73%	6.87 (1.29×)	1.22 (55.36×)	**8**.09 (**9**.45×)	172.69 (1.32×)	0.32 (530.92×)	173.01 (2.30×)
ASNet(9)	**72.00%**	**90.61%**	13.39	2.07 (30.49×)	15.46 (4.95×)	247.14	0.42 (357.10×)	247.56 (1.61×)
ASNet-s(9)	71.38%	90.28%	9.38 (1.43×)	1.94 (32.49×)	11.32 (6.75×)	183.27 (1.35×)	0.51 (296.74×)	183.78 (2.17×)

like a trouser for a human. However, the deep neural network misclassifies it as a t-shirt/top.

6.5.2.2 CIFAR-10

Next, we show the numerical results of attacking VGG-19 on CIFAR-10. Fig. 6.8 compares the active-subspace method compared with the baseline UAP and Gaussian random vectors. The top figures show the results by the dataset in the first class (i.e., automobile), and the bottom figures show the results for all ten classes. For both two cases, the proposed active-subspace attack can achieve 20% higher attack ratios while three times faster than UAP. This is similar to the results in Fashion-MNIST because the active subspace has a better ability to capture global information.

We further show the effects of *different number of training samples* in Fig. 6.9. When the number of samples is increased, the testing attack ratio gets better. In our numerical experiments, we set the number of samples as 100 for one-class experiments and 200 for all-classes experiments.

We continue to show the *cross-model* performance on four different ResNet networks and one VGG network. We test the performance of the attack vector trained from one model on all other models. Each row in Table 6.6 shows the results on the same deep neural network, and each column shows the results of the same attack vector. It shows that ResNet-20 is easier to be attacked compared with other models. This agrees with our intuition that a simple network structure such as ResNet-20 is less robust. On the contrary, VGG-19 is the most robust. The success of cross-model attacks indicates that these neural networks could find a similar feature.

TABLE 6.5: Accuracy and storage on ResNet-110 for CIFAR-100. Here, "Pre-M" denotes the pre-model, i.e., layers 1 to l of the original deep neural networks, "AS" and "PCE" denote the active subspace and polynomial chaos expansion layer, respectively.

Network	**Top-1**	**Top-5**	**Storage (MB)**			**Flops (10^6)**		
ResNet-110	71.94%	91.71 %	6.61			252.89		
			Pre-M	**AS+PCE**	**Overall**	**Pre-M**	**AS+PCE**	**Overall**
ASNet(75)	63.01%	88.55%	1.79	1.29 (3.73×)	3.08 (2.14×)	172.67	0.22 (367.88×)	172.89 (1.46×)
ASNet-s(75)	63.16%	88.65%	1.47 (1.22×)	1.20 (3.99×)	**2.67** (2.46×)	143.11 (1.21×)	0.31 (254.69×)	**143.42** (1.76×)
ASNet(81)	65.82%	90.02%	2.64	1.29 (3.07×)	3.93 (1.68×)	186.83	0.22 (302.96×)	187.04 (1.35×)
ASNet-s(81)	65.73%	89.95%	2.20 (1.20×)	1.21 (3.27×)	3.41 (1.93×)	155.61 (1.20×)	0.32 (208.38×)	155.93 (1.62×)
ASNet(87)	**67.71%**	**90.17%**	3.48	1.29 (2.41×)	4.77 (1.38×)	200.98	0.22 (238.04×)	201.20 (1.26×)
ASNet-s(87)	67.65%	90.10%	2.91 (1.20×)	1.21 (2.56×)	4.12 (1.60×)	166.50 (1.21×)	0.32 (163.50×)	166.81 (1.52×)

TABLE 6.6: Cross-model performance for CIFAR-10

	ResNet-20	ResNet-44	ResNet-56	ResNet-110	VGG-19
ResNet-20	**91.35%**	87.74%	86.28%	87.38%	81.16%
ResNet-44	84.75%	**92.28%**	87.03%	85.44%	83.44%
ResNet-56	83.63%	86.67%	**90.15%**	87.39%	84.38%
ResNet-110	71.02%	77.58%	74.19%	**92.77%**	77.32%
VGG-19	53.61%	59.74%	61.49%	66.29%	**80.02%**

6.5.2.3 CIFAR-100

Finally, we show the results on CIFAR-100 for both the first class (i.e., dolphin) and all classes. Similar to Fashion-MNIST and CIFAR-10, Fig. 6.10 shows that active-subspace can achieve higher attack ratios than both UAP and Gaussian random vectors. Further, compared with CIFAR-10, CIFAR-100 is easier to be attacked partially because it has more classes.

We summarize the results for different datasets in Table 6.7. The second column shows the number of classes in the dataset. In terms of testing attack ratio for the whole dataset, active-subspace achieves 24.2%, 15%, and 6.1% higher attack ratios than UAP for Fashion-MNIST, CIFAR-10, and CIFAR-100, respectively. In terms of the CPU time, active-subspace achieves 42×, 5×, and 14× speedup than UAP on the Fashion-MNIST, CIFAR-10, and CIFAR-100, respectively.

6.6 Conclusions and Discussions

This chapter has analyzed deep neural networks by the active subspace method originally developed for dimensionality reduction of uncertainty quantification. We have investigated two problems: how many neurons and

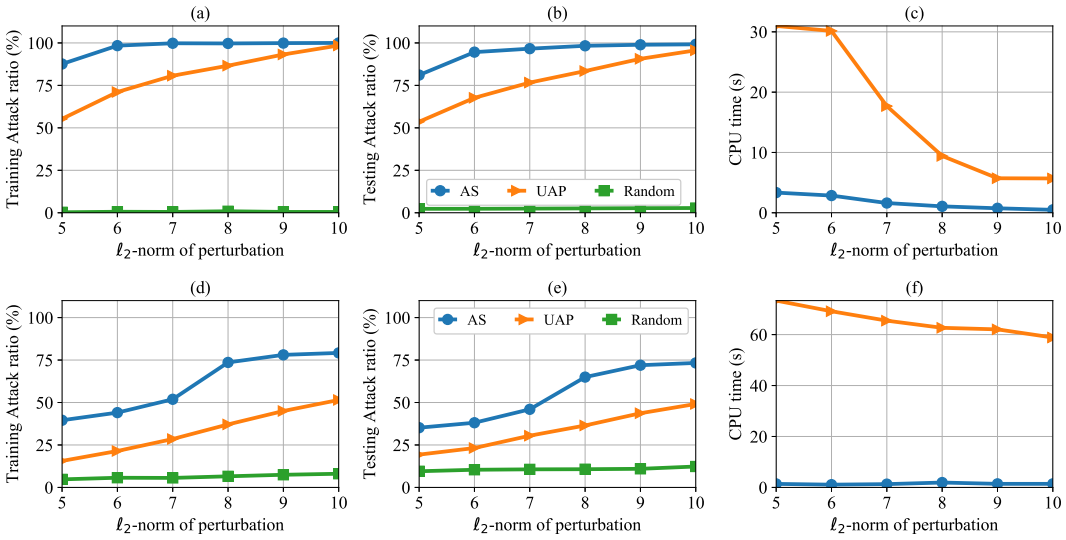


FIGURE 6.6: Universal adversarial attacks for the Fashion-MINST with respect to different ℓ_2-norms. (a)-(c): the results for attacking one class dataset. (d)-(f): the results for attacking the whole dataset.

TABLE 6.7: Summary of the universal attack for different datasets by the active-subspace compared with UAP and the random vector. The norm of perturbation is equal to 10.

		Training Attack ratio			Testing Attack ratio			CPU time (s)	
	# Class	AS	UAP	Rand	AS	UAP	Rand	AS	UAP
Fashion-MNIST	1	100.0%	93.6%	1.8%	**98.0%**	91.3%	3.0%	**0.15**	5.49
	10	79.2%	51.5%	8.0%	**73.3%**	49.1%	12.3%	**1.40**	58.85
CIFAR-10	1	94.7%	79.8%	8.0%	**84.5%**	57.9%	10.6%	**8.18**	52.83
	10	86.5%	65.9%	10.2%	**74.9%**	59.9%	17.0%	**37.01**	181.72
CIFAR-100	1	97.2%	87.9%	19.7%	**92.1%**	84.3%	37.9%	**13.32**	248.78
	100	93.7%	86.5%	38.7%	**83.5%**	77.4%	52.0%	**14.32**	204.50

layers are necessary (or important) in a deep neural network, and how to generate a universal adversarial attack vector that can be applied to a set of testing data? Firstly, we have presented a definition of "the number of active neurons" and have shown its theoretical error bounds for model reduction. Our numerical study has shown that many neurons and layers are not needed. Based on this observation, we have proposed a new network called ASNet by cutting off the whole neural network at a proper layer and replacing all subsequent layers with an active subspace layer and a polynomial chaos expansion layer. The numerical experiments show that the proposed deep neural network structural analysis method can produce a new network with significant storage savings and computational speedup yet with little accuracy loss. Our methods can be combined with existing model compression techniques (e.g., pruning, quantization and low-rank factorization) to develop compact deep neural network models that are more suitable for the deployment on resource-constrained platforms. Secondly, we have applied the active subspace to generate a universal attack vector that is independent of a specific

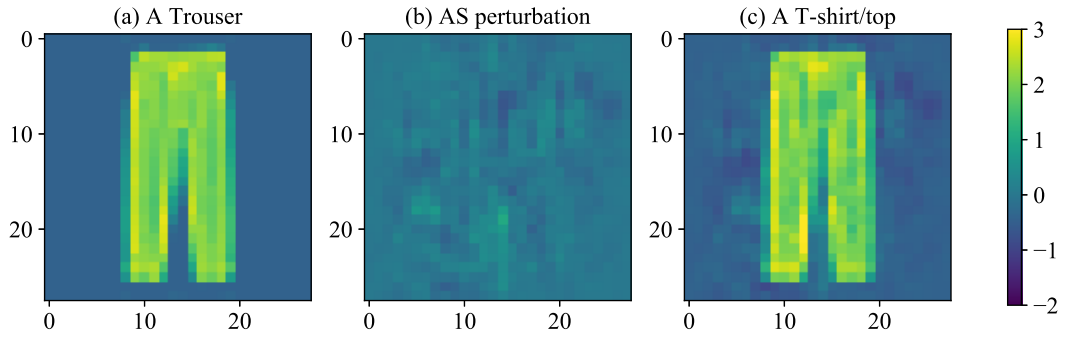


FIGURE 6.7: The effect of our attack method on one data sample in the Fashion-MNIST dataset. (a) A trouser from the original dataset. (b) An active-subspace perturbation vector with the ℓ_2 norm equals 5. (c) The perturbed sample is misclassified as a t-shirt/top by the deep neural network.

data sample and can be applied to a whole dataset. Our proposed method can achieve a much higher attack ratio than the existing work [120] and enjoys a lower computational cost.

ASNet has two main goals: to detect the necessary neurons and layers, and to compress the existing network. To fulfill the first goal, we require a pre-trained model because from Lemmas 6.3.1, and 6.3.2, the accuracy of the reduced model will approach that of the original one. For the second task, the pre-trained model helps us to get a good estimation for the number of active neurons, a proper layer to cut off, and a good initialization for the active subspace layer and polynomial chaos expansion layer. However, a pre-trained model is not required because we can construct ASNet in a heuristic way (as done in most DNN): a reasonable guess for the number of active neurons and cut-off layer, and a random parameter initialization for the pre-model, the active subspace layer and the polynomial chaos expansion layer.

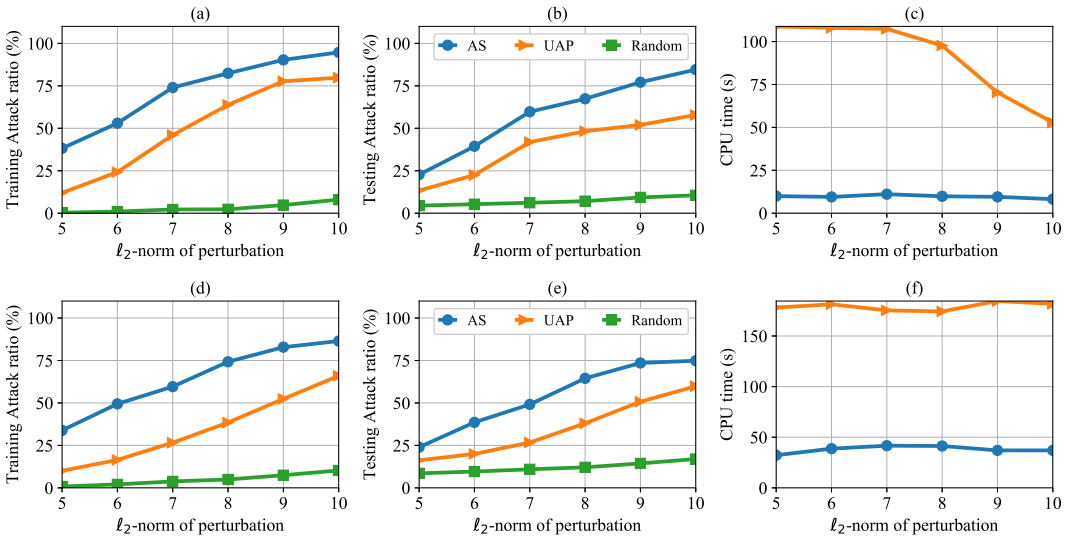


FIGURE 6.8: Universal adversarial attacks of VGG-19 on CIFAR-10 with respect to different ℓ_2-norm perturbations. (a)-(c): The training attack ratio, the testing attack ratio, and the CPU time in seconds for attacking one class dataset. (d)-(f): The results for attacking ten classes dataset together.

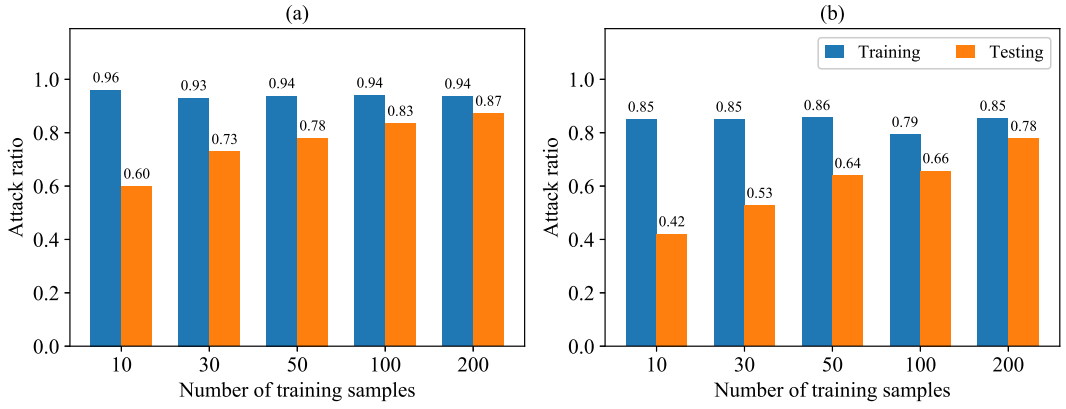


FIGURE 6.9: Adversarial attack of VGG-19 on CIFAR-10 with different number of training samples. The ℓ_2-norm perturbation is fixed as 10. (a) The results of attacking the dataset from the first class; (b) The results of attacking the whole dataset with 10 classes.

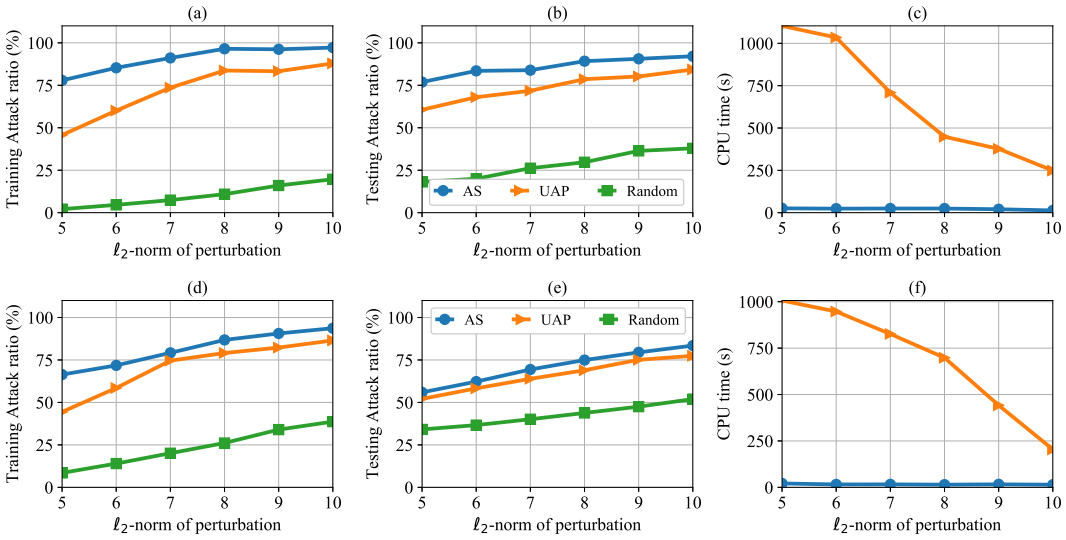


FIGURE 6.10: Results for universal adversarial attack for CIFAR-100 with respect to different ℓ_2-norm perturbations. (a)-(c): The results for attacking the dataset from the first class. (d)-(f): The results for attacking ten classes dataset together.

Conclusions

In this thesis, we focused on the investigation and advancement of neural ordinary differential equations (ODEs) as a tool for modeling continuous-time dynamic systems. Our research efforts were divided into two interconnected parts, with the first part dedicated to the study of various aspects of neural ODEs.

In the first part, we developed an efficient training algorithm for neural ODEs in chapter 2, which has the potential to significantly improve the performance of neural ODEs in a variety of applications. Additionally, we conducted a thorough experimental evaluation of different normalization techniques for neural ODEs in chapter 3, as normalization is an important aspect of neural network training that can impact the effectiveness of neural ODEs. Finally, in chapter 4, we proposed a method for training neural ODEs that are more robust to adversarial attacks, addressing a major concern in machine learning and offering the potential for significant progress in this area.

In the second part of this thesis, we applied classical model reduction techniques, namely reduced-order modeling (chapter 5) and the active subspace method (chapter 6), to neural networks.

Bibliography

[1] Sajjad Abdoli et al. "Universal Adversarial Audio Perturbations". In: *arXiv preprint arXiv:1908.03173* (2019).

[2] Alireza Aghasi et al. "Net-trim: Convex pruning of deep neural networks with performance guarantee". In: *Advances in Neural Information Processing Systems*. 2017, pp. 3177–3186.

[3] Naveed Akhtar and Ajmal Mian. "Threat of adversarial attacks on deep learning in computer vision: A survey". In: *IEEE Access* 6 (2018), pp. 14410–14430.

[4] Shun-ichi Amari, Andrzej Cichocki, and Howard Yang. "A new learning algorithm for blind signal separation". In: *Advances in neural information processing systems* 8 (1995).

[5] Harbir Antil and Dmitriy Leykekhman. "A brief introduction to PDE-constrained optimization". In: *Frontiers in PDE-constrained optimization*. Springer, 2018, pp. 3–40.

[6] Larry Armijo. "Minimization of functions having Lipschitz continuous first partial derivatives". In: *Pacific Journal of mathematics* 16.1 (1966), pp. 1–3.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[8] Shumeet Baluja and Ian Fischer. "Adversarial transformation networks: Learning to generate adversarial examples". In: *arXiv preprint arXiv:1703.09387* (2017).

[9] Melika Behjati et al. "Universal Adversarial Attacks on Text Classifiers". In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 7345–7349.

[10] Jean-Paul Berrut and Lloyd N Trefethen. "Barycentric Lagrange interpolation". In: *SIAM Review* 46.3 (2004), pp. 501–517.

[11] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[12] Ernesto G Birgin, José Mario Martínez, and Marcos Raydan. "Nonmonotone spectral projected gradient methods on convex sets". In: *SIAM Journal on Optimization* 10.4 (2000), pp. 1196–1211.

[13] Andrew M Bradley. *PDE-constrained optimization and the adjoint method*. Tech. rep. Technical Report. Stanford University. https://cs. stanford. edu/~ ambrad . . ., 2013.

[14] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. "Model Compression". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: ACM, 2006, pp. 535–541. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150464. URL: http://doi.acm.org/10.1145/1150402.1150464.

[15] John Charles Butcher and Nicolette Goodwin. *Numerical methods for ordinary differential equations*. Vol. 2. Wiley Online Library, 2008.

[16] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *arXiv preprint arXiv:1812.00332* (2018).

[17] Nicholas Carlini and David Wagner. "Towards evaluating the robustness of neural networks". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 39–57.

[18] Yair Carmon et al. "Unlabeled data improves adversarial robustness". In: *arXiv preprint arXiv:1905.13736* (2019).

[19] Fabio Carrara et al. "On the robustness to adversarial examples of neural ode image classifiers". In: *2019 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE. 2019, pp. 1–6.

[20] Bo Chang et al. "Multi-level residual networks from dynamical systems view". In: *arXiv preprint arXiv:1710.10348* (2017).

[21] Saifon Chaturantabut and Danny C Sorensen. "Nonlinear model reduction via discrete empirical interpolation". In: *SIAM Journal on Scientific Computing* 32.5 (2010), pp. 2737–2764.

[22] Tian Qi Chen et al. "Neural ordinary differential equations". In: *Advances in Neural Information Processing Systems*. 2018, pp. 6571–6583.

[23] Yu Cheng et al. "Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136. ISSN: 10535888. DOI: 10.1109/MSP.2017.2765695. arXiv: 1710.09282.

[24] Earl A Coddington and Norman Levinson. *Theory of ordinary differential equations*. Tata McGraw-Hill Education, 1955.

[25] Paul G Constantine. *Active subspaces: Emerging ideas for dimension reduction in parameter studies*. Vol. 2. SIAM, 2015.

[26] Paul G Constantine and Alireza Doostan. "Time-dependent global sensitivity analysis with active subspaces for a lithium ion battery model". In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 10.5 (2017), pp. 243–262.

[27] Paul G Constantine, Eric Dow, and Qiqi Wang. "Active subspace methods in theory and practice: applications to kriging surfaces". In: *SIAM Journal on Scientific Computing* 36.4 (2014), A1500–A1524.

[28] Paul G Constantine et al. "Exploiting active subspaces to quantify uncertainty in the numerical simulation of the HyShot II scramjet". In: *Journal of Computational Physics* 302 (2015), pp. 1–20.

[29] Tim Cooijmans et al. "Recurrent batch normalization". In: *arXiv preprint arXiv:1603.09025* (2016).

[30] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Training deep neural networks with low precision multiplications". In: *arXiv preprint arXiv:1412.7024* (2014). arXiv: 1412.7024. URL: http://arxiv.org/abs/1412.7024.

[31] Matthieu Courbariaux et al. "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or-1". In: *arXiv preprint arXiv:1602.02830* (2016).

[32] Francesco Croce et al. "RobustBench: a standardized adversarial robustness benchmark". In: *arXiv preprint arXiv:2010.09670* (2020).

[33] Chunfeng Cui and Zheng Zhang. "High-Dimensional Uncertainty Quantification of Electronic and Photonic IC with Non-Gaussian Correlated Process Variations". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[34] Chunfeng Cui and Zheng Zhang. "Stochastic collocation with non-gaussian correlated process variations: Theory, algorithms and applications". In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* (2018).

[35] Chunfeng Cui et al. "Active Subspace of Neural Networks: Structural Analysis and Universal Attacks". In: *arXiv preprint arXiv:1910.13025* (2019).

[36] Talgat Daulbaev et al. "Interpolation Technique to Speed Up Gradients Propagation in Neural ODEs". In: *Advances in Neural Information Processing Systems* 33 (2020).

[37] Nicola Demo, Marco Tezzele, and Gianluigi Rozza. "A non-intrusive approach for the reconstruction of POD modal coefficients through active subspaces". In: *Comptes Rendus Mécanique* 347.11 (2019), pp. 873–881.

[38] Lei Deng et al. "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework". In: *Neural Networks* 100 (2018), pp. 49–58.

[39] Emily L Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems*. 2014, pp. 1269–1277.

[40] John R Dormand and Peter J Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26.

[41] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. "Augmented neural ODEs". In: *arXiv preprint arXiv:1904.01681* (2019).

[42] Gintare Karolina Dziugaite, Zoubin Ghahramani, and Daniel M Roy. "A study of the effect of jpg compression on adversarial images". In: *arXiv preprint arXiv:1608.00853* (2016).

[43] Radii Petrovich Fedorenko. "A relaxation method for solving elliptic difference equations". In: *USSR Computational Mathematics and Mathematical Physics* 1.4 (1962), pp. 1092–1096.

[44] Andreas Fichtner, H-P Bunge, and Heiner Igel. "The adjoint method in seismology: I. Theory". In: *Physics of the Earth and Planetary Interiors* 157.1-2 (2006), pp. 86–104.

[45] Gerald B Folland. *Introduction to Partial Differential Equations*. Princeton university press, 1995.

[46] Alexander Fonarev et al. "Efficient rectangular maximal-volume algorithm for rating elicitation in collaborative filtering". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 141–150.

[47] Bengt Fornberg. *A Practical Guide to Pseudospectral Methods*. Vol. 1. Cambridge university press, 1998.

[48] Jonathan Frankle and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635* (2018).

[49] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *arXiv preprint arXiv:1704.04861* (Apr. 2017).

[50] Xitong Gao et al. "Dynamic Channel Pruning: Feature Boosting and Suppression". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=BJxh2j0qYm.

[51] Timur Garipov et al. "Ultimate tensorization: compressing convolutional and FC layers alike". In: *arXiv preprint arXiv:1611.03214* (2016).

[52] Rong Ge, Runzhe Wang, and Haoyu Zhao. "Mildly Overparametrized Neural Nets can Memorize Training Data Efficiently". In: *arXiv preprint arXiv:1909.11837* (2019).

[53] Roger G Ghanem and Pol D Spanos. "Stochastic Finite Element Method: Response Statistics". In: *Stochastic Finite Elements: A Spectral Approach*. Springer, 1991, pp. 101–119.

[54] Mina Ghashami et al. "Frequent directions: Simple and deterministic matrix sketching". In: *SIAM Journal on Computing* 45.5 (2016), pp. 1762–1792.

[55] Amir Gholami, Kurt Keutzer, and George Biros. "ANODE: Unconditionally Accurate Memory-Efficient Gradients for Neural ODEs". In: *arXiv preprint arXiv:1902.10298* (2019).

[56] Arnab Ghosh et al. "STEER: Simple Temporal Regularization For Neural ODEs". In: *arXiv preprint arXiv:2006.10711* (2020).

[57] Yotam Gil et al. "White-to-Black: Efficient Distillation of Black-Box Adversarial Attacks". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 1373–1379.

[58] Michael B Giles and Niles A Pierce. "An introduction to the adjoint approach to design". In: *Flow, Turbulence and Combustion* 65.3-4 (2000), pp. 393–415.

[59] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples". In: *arXiv preprint arXiv:1412.6572* (2014).

[60] Sergei A Goreinov et al. "How to find a good submatrix". In: *Matrix Methods: Theory, Algorithms And Applications: Dedicated to the Memory of Gene Golub*. World Scientific, 2010, pp. 247–256.

[61] Will Grathwohl et al. "Ffjord: Free-form continuous dynamics for scalable reversible generative models". In: *arXiv preprint arXiv:1810.01367* (2018).

[62] Alex Graves et al. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 369–376.

[63] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: *International Conference on Machine Learning*. 2015, pp. 1737–1746.

[64] Julia Gusak et al. "Automated Multi-Stage Compression of Neural Networks". In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 2019, pp. 0–0.

[65] Julia Gusak et al. "Reduced-order modeling of deep neural networks". In: *Computational Mathematics and Mathematical Physics* 61.5 (2021), pp. 774–785.

[66] Julia Gusak et al. "Towards Understanding Normalization in Neural ODEs". In: *International Conference on Learning Representations (ICLR) Workshop on Integration of Deep Neural Models and Differential Equations* (2020). URL: https://openreview.net/forum?id=mllQ3QNNr9d.

[67] Ernst Hairer, Syvert P Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer-Vlg, 1993.

[68] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM review* 53.2 (2011), pp. 217–288.

[69] Matthew CG Hall. "Application of adjoint sensitivity theory to an atmospheric general circulation model". In: *Journal of the Atmospheric Sciences* 43.22 (1986), pp. 2644–2652.

[70] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[71] YAN Hanshu et al. "On robustness of neural ordinary differential equations". In: *International Conference on Learning Representations*. 2019.

[72] Cole Hawkins and Zheng Zhang. "Bayesian Tensorized Neural Networks with Automatic Rank Selection". In: *arXiv preprint arXiv:1905.10478* (2019).

[73] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[74] Kaiming He et al. "Identity mappings in deep residual networks". In: *European conference on computer vision*. Springer. 2016, pp. 630–645.

[75] Yihui He, Xiangyu Zhang, and Jian Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". In: *The IEEE International Conference on Computer Vision (ICCV)*. 2017.

[76] Yihui He et al. "Amc: Automl for model compression and acceleration on mobile devices". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 784–800.

[77] Nicholas J Higham. "The numerical stability of barycentric Lagrange interpolation". In: *IMA Journal of Numerical Analysis* 24.4 (2004), pp. 547–556.

[78] Alan C Hindmarsh et al. "User Documentation for ida v5. 4.0 (sundials v5. 4.0)". In: (2020).

[79] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: *arXiv preprint arXiv:1503.02531* (2015).

[80] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: *stat* 1050 (2015), p. 9.

[81] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[82] Wassily Hoeffding. "Probability inequalities for sums of bounded random variables". In: *The Collected Works of Wassily Hoeffding*. Springer, 1994, pp. 409–426.

[83] Mary Kathleen Horn. "Fourth-and fifth-order, scaled rungs–kutta algorithms for treating dense output". In: *SIAM Journal on Numerical Analysis* 20.3 (1983), pp. 558–568.

[84] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. Vol. 398. John Wiley & Sons, 2013.

[85] Hengyuan Hu et al. "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures". In: *arXiv preprint arXiv:1607.03250* (2016).

[86] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[87] Michael F Hutchinson. "A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines". In: *Communications in Statistics-Simulation and Computation* 18.3 (1989), pp. 1059–1076.

[88] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[89] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. "Speeding up Convolutional Neural Networks with Low Rank Expansions". In: *arXiv preprint arXiv:1405.3866* (2014). arXiv: 1405.3866. URL: http://arxiv.org/abs/1405.3866.

[90] Ian Jolliffe. "Principal component analysis". In: *International encyclopedia of statistical science*. Springer, 2011, pp. 1094–1096.

[91] Can Kanbak, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. "Geometric robustness of deep networks: analysis and improvement". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4441–4449.

[92] Jacob Kelly et al. "Learning Differential Equations that are Easy to Solve". In: *arXiv preprint arXiv:2007.04504* (2020).

[93] Valentin Khrulkov and Ivan Oseledets. "Art of singular vectors and universal adversarial perturbations". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8562–8570.

[94] Valentin Khrulkov and Ivan Oseledets. "Understanding DDPM latent codes through optimal transport". In: *arXiv preprint arXiv:2202.07477* (2022).

[95] Patrick Kidger et al. "Neural controlled differential equations for irregular time series". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6696–6707.

[96] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[97] Diederik P Kingma and Max Welling. "Stochastic gradient VB and the variational auto-encoder". In: *Second International Conference on Learning Representations, ICLR*. Vol. 19. 2014.

[98] Durk P Kingma and Prafulla Dhariwal. "Glow: Generative flow with invertible 1x1 convolutions". In: *Advances in Neural Information Processing Systems*. 2018, pp. 10215–10224.

[99] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[100] Karl Kunisch and Stefan Volkwein. "Galerkin proper orthogonal decomposition methods for a general equation in fluid dynamics". In: *SIAM Journal on Numerical analysis* 40.2 (2002), pp. 492–515.

[101] Vadim Lebedev et al. "Speeding-up convolutional neural networks using fine-tuned cp-decomposition". In: *arXiv preprint arXiv:1412.6553* (2014).

[102] Hao Li et al. "Pruning filters for efficient convnets". In: *arXiv preprint arXiv:1608.08710* (2016).

[103] Xuechen Li et al. "Scalable gradients for stochastic differential equations". In: *arXiv preprint arXiv:2001.01328* (2020).

[104] David R Lide. "Handbook of mathematical functions". In: *A Century of Excellence in Measurements, Standards, and Technology*. CRC Press, 2018, pp. 135–139.

[105] Liu Liu et al. "Dynamic Sparse Graph for Efficient Deep Learning". In: *arXiv preprint arXiv:1810.00859* (2018).

[106] Xuanqing Liu et al. "Neural SDE: Stabilizing neural ODE networks with stochastic noise". In: *arXiv preprint arXiv:1906.02355* (2019).

[107] Zhuang Liu et al. "Learning Efficient Convolutional Networks through Network Slimming". In: *ICCV*. 2017.

[108] Zhuang Liu et al. "Rethinking the value of network pruning". In: *arXiv preprint arXiv:1810.05270* (2018).

[109] Tyson Loudon and Stephen Pankavich. "Mathematical analysis and dynamic active subspaces for a long term model of HIV". In: *arXiv preprint arXiv:1604.04588* (2016).

[110] Yiping Lu et al. "Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations". In: *arXiv preprint arXiv:1710.10121* (2017).

[111] J. Luo et al. "ThiNet: Pruning CNN Filters for a Thinner Net". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018), pp. 1–1. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2018.2858232.

[112] Ping Luo et al. "Differentiable learning-to-normalize via switchable normalization". In: *arXiv preprint arXiv:1806.10779* (2018).

[113] Aleksander Madry et al. "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *International Conference on Learning Representations*. 2018.

[114] Guri I Marchuk. *Adjoint Equations and Analysis of Complex Systems*. Vol. 295. Springer Science & Business Media, 2013.

[115] Daniil Merkulov and Ivan Oseledets. "Stochastic gradient algorithms from ODE splitting perspective". In: *arXiv preprint arXiv:2004.08981* (2020).

[116] Aleksandr Mikhalev and Ivan V Oseledets. "Rectangular maximum-volume submatrices and their applications". In: *Linear Algebra and its Applications* 538 (2018), pp. 187–211.

[117] Takeru Miyato et al. "Spectral Normalization for Generative Adversarial Networks". In: *arXiv preprint arXiv:1802.05957* (2018).

[118] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. "Variational dropout sparsifies deep neural networks". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2498–2507.

[119] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. "Deepfool: a simple and accurate method to fool deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2574–2582.

[120] Seyed-Mohsen Moosavi-Dezfooli et al. "Universal adversarial perturbations". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1765–1773.

[121] Shinichi Nakajima et al. "Global analytic solution of fully-observed variational Bayesian matrix factorization". In: *Journal of Machine Learning Research* 14.Jan (2013), pp. 1–37.

[122] Paarth Neekhara et al. "Universal adversarial perturbations for speech recognition systems". In: *arXiv preprint arXiv:1905.03828* (2019).

[123] Tan M Nguyen et al. "InfoCNF: An efficient conditional continuous normalizing flow with adaptive solvers". In: *arXiv preprint arXiv:1912.03978* (2019).

[124] Alexander Novikov et al. "Tensorizing neural networks". In: *Advances in Neural Information Processing Systems*. 2015, pp. 442–450.

[125] Viktor Oganesyan, Alexandra Volokhova, and Dmitry Vetrov. "Stochasticity in Neural ODEs: An Empirical Study". In: *arXiv preprint arXiv:2002.09779* (2020).

[126] Katharina Ott et al. "When are Neural ODE Solutions Proper ODEs?" In: *arXiv preprint arXiv:2007.15386* (2020).

[127] Samet Oymak and Mahdi Soltanolkotabi. "Towards moderate overparameterization: global convergence guarantees for training shallow neural networks". In: *arXiv preprint arXiv:1902.04674* (2019).

[128] Utku Ozbulak et al. "Perturbation analysis of gradient-based adversarial attacks". In: *Pattern Recognition Letters* 135 (2020), pp. 313–320.

[129] Nicolas Papernot et al. "The limitations of deep learning in adversarial settings". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 372–387.

[130] Sunghyun Park et al. "Vid-ode: Continuous-time video generation with neural ordinary differential equation". In: *arXiv preprint arXiv:2010.08188* (2020).

[131] R-E Plessix. "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications". In: *Geophysical Journal International* 167.2 (2006), pp. 495–503.

[132] LS Pontryagin et al. *Mathematical Theory of Optimal Processes {in Russian}*. 1961.

[133] Alessio Quaglino et al. "SNODE: Spectral Discretization of Neural ODEs for System Identification". In: *arXiv preprint arXiv:1906.07038* (2019).

[134] Alfio Quarteroni, Gianluigi Rozza, et al. *Reduced order methods for modeling and computational reduction*. Vol. 9. Springer, 2014.

[135] Alejandro F Queiruga et al. "Continuous-in-Depth Neural Networks". In: *arXiv preprint arXiv:2008.02389* (2020).

[136] Christopher Rackauckas et al. "Universal differential equations for scientific machine learning". In: *arXiv preprint arXiv:2001.04385* (2020).

[137] Jonas Rauber, Wieland Brendel, and Matthias Bethge. "Foolbox: A Python toolbox to benchmark the robustness of machine learning models". In: *Reliable Machine Learning in the Wild Workshop, 34th International Conference on Machine Learning*. 2017. URL: http://arxiv.org/abs/1707.04131.

[138] Jonas Rauber et al. "Foolbox Native: Fast adversarial attacks to benchmark the robustness of machine learning models in PyTorch, TensorFlow, and JAX". In: *Journal of Open Source Software* 5.53 (2020), p. 2607. DOI: 10.21105/joss.02607. URL: https://doi.org/10.21105/joss.02607.

[139] Danilo Rezende and Shakir Mohamed. "Variational inference with normalizing flows". In: *International conference on machine learning*. PMLR. 2015, pp. 1530–1538.

[140] Byron P Roe et al. "Boosted decision trees as an alternative to artificial neural networks for particle identification". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 543.2-3 (2005), pp. 577–584.

[141] Adriana Romero et al. "Fitnets: Hints for thin deep nets". In: *arXiv preprint arXiv:1412.6550* (2014).

[142] Clarence W Rowley, Tim Colonius, and Richard M Murray. "Model reduction for compressible flows using POD and Galerkin projection". In: *Physica D: Nonlinear Phenomena* 189.1-2 (2004), pp. 115–129.

[143] Halsey Lawrence Royden. *Real Analysis*. Macmillan, 2010.

[144] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. "Latent Ordinary Differential Equations for Irregularly-Sampled Time Series". In: *Advances in Neural Information Processing Systems*. 2019, pp. 5321–5331.

[145] Trent Michael Russi. "Uncertainty quantification with experimental data and complex system models". PhD thesis. UC Berkeley, 2010.

[146] Lars Ruthotto and Eldad Haber. "Deep neural networks motivated by partial differential equations". In: *Journal of Mathematical Imaging and Vision* (2018), pp. 1–13.

[147] Tara N Sainath et al. "Low-rank matrix factorization for deep neural network training with high-dimensional output targets". In: *IEEE international conference on acoustics, speech and signal processing*. 2013, pp. 6655–6659.

[148] Tim Salimans and Durk P Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks". In: *Advances in neural information processing systems*. 2016, pp. 901–909.

[149] Shibani Santurkar et al. "How does batch normalization help optimization?" In: *Advances in Neural Information Processing Systems*. 2018, pp. 2483–2493.

[150] Simone Scardapane et al. "Group sparse regularization for deep neural networks". In: *Neurocomputing* 241 (2017), pp. 81–89.

[151] Mark Schmidt et al. "Optimizing costly functions with simple constraints: A limited-memory projected quasi-newton algorithm". In: *Artificial Intelligence and Statistics*. 2009, pp. 456–463.

[152] Vikash Sehwag et al. "Improving Adversarial Robustness Using Proxy Distributions". In: *arXiv preprint arXiv:2104.09425* (2021).

[153] Alexandru Constantin Serban and Erik Poll. "Adversarial examples-a complete characterisation of the phenomenon". In: *arXiv preprint arXiv:1810.01185* (2018).

[154] Radu Serban and Alan C Hindmarsh. "CVODES: the sensitivity-enabled ODE solver in SUNDIALS". In: *ASME 2005 international design engineering technical conferences and computers and information in engineering conference*. American Society of Mechanical Engineers Digital Collection. 2005, pp. 257–269.

[155] Shai Shalev-Shwartz and Tong Zhang. "Accelerated proximal stochastic dual coordinate ascent for regularized loss minimization". In: *International Conference on Machine Learning*. 2014, pp. 64–72.

[156] Lawrence F Shampine. "Interpolation for Runge–Kutta methods". In: *SIAM journal on Numerical Analysis* 22.5 (1985), pp. 1014–1027.

[157] Lawrence F Shampine. "Some practical Runge–Kutta formulas". In: *Mathematics of Computation* 46.173 (1986), pp. 135–150.

[158] Hidetoshi Shimodaira. "Improving predictive inference under covariate shift by weighting the log-likelihood function". In: *Journal of statistical planning and inference* 90.2 (2000), pp. 227–244.

[159] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[160] Gustaf Söderlind. "The logarithmic norm. History and modern theory". In: *BIT Numerical Mathematics* 46.3 (2006), pp. 631–652.

[161] Gustaf Söderlind, Laurent Jay, and Manuel Calvo. "Stiffness 1952–2012: Sixty years in search of a definition". In: *BIT Numerical Mathematics* 55.2 (2015), pp. 531–558.

[162] Gustaf Söderlind and Robert MM Mattheij. "Stability and asymptotic estimates in nonautonomous linear differential systems". In: *SIAM Journal on Mathematical Analysis* 16.1 (1985), pp. 69–92.

[163] Yang Song et al. "Maximum likelihood training of score-based diffusion models". In: *Advances in Neural Information Processing Systems* 34 (2021).

[164] Christian Szegedy et al. "Intriguing properties of neural networks". In: *arXiv preprint arXiv:1312.6199* (2013).

[165] Lloyd N Trefethen. *Spectral Methods in MATLAB*. Vol. 10. Siam, 2000.

[166] Anton Tsitsulin et al. "FREDE: Linear-Space Anytime Graph Embeddings". In: *arXiv preprint arXiv:2006.04746* (2020).

[167] Belinda Tzen and Maxim Raginsky. "Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit". In: *arXiv preprint arXiv:1905.09883* (2019).

[168] Panayot S Vassilevski. *Lecture notes on multigrid methods*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2010.

[169] Gerhard Wanner and Ernst Hairer. *Solving Ordinary Differential Equations II*. Springer Berlin Heidelberg, 1996.

[170] Wei Wen et al. "Learning structured sparsity in deep neural networks". In: *Advances in neural information processing systems*. 2016, pp. 2074–2082.

[171] Eric Wong, Leslie Rice, and J Zico Kolter. "Fast is better than free: Revisiting adversarial training". In: *arXiv preprint arXiv:2001.03994* (2020).

[172] David P Woodruff et al. "Sketching as a tool for numerical linear algebra". In: *Foundations and Trends® in Theoretical Computer Science* 10.1–2 (2014), pp. 1–157.

[173] Lei Wu and Zhanxing Zhu. "Towards Understanding and Improving the Transferability of Adversarial Examples in Deep Neural Networks". In: *Asian Conference on Machine Learning*. PMLR. 2020, pp. 837–850.

[174] Dongbin Xiu and George Em Karniadakis. "Modeling uncertainty in steady state diffusion problems via generalized polynomial chaos". In: *Computer methods in applied mechanics and engineering* 191.43 (2002), pp. 4927–4948.

[175] Dongbin Xiu and George Em Karniadakis. "The Wiener–Askey polynomial chaos for stochastic differential equations". In: *SIAM journal on scientific computing* 24.2 (2002), pp. 619–644.

[176] Shaokai Ye et al. "Progressive DNN Compression: A Key to Achieve Ultra-High Weight Pruning and Quantization Rates using ADMM". In: *arXiv preprint arXiv:1903.09769* (2019).

[177] Tom Young et al. "Recent trends in deep learning based natural language processing". In: *ieee Computational intelligenCe magazine* 13.3 (2018), pp. 55–75.

[178] Igor Zacharov et al. ""Zhores"—Petaflops supercomputer for data-driven modeling, machine learning and artificial intelligence installed in Skolkovo Institute of Science and Technology". In: *Open Engineering* 9.1 (2019), pp. 512–520.

[179] Sergey Zagoruyko and Nikos Komodakis. "Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer". In: *arXiv preprint arXiv:1612.03928* (2016). arXiv: 1612.03928. URL: http://arxiv.org/abs/1612.03928.

[180] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[181] Vitaly P Zankin, Gleb V Ryzhakov, and Ivan Oseledets. "Gradient Descent-based D-optimal Design for the Least-Squares Polynomial Approximation". In: *arXiv preprint arXiv:1806.06631* (2018).

[182] Tianjun Zhang et al. "ANODEV2: A Coupled Neural ODE Evolution Framework". In: *arXiv preprint arXiv:1906.04596* (2019).

[183] Xiangyu Zhang et al. "Accelerating very deep convolutional networks for classification and detection". In: *IEEE transactions on pattern analysis and machine intelligence* 38.10 (2015), pp. 1943–1955.

[184] Jing Zhong et al. "Where to Prune: Using LSTM to Guide End-to-end Pruning." In: *IJCAI*. 2018, pp. 3205–3211.

[185] Juntang Zhuang et al. "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE". In: *arXiv preprint arXiv:2006.02493* (2020).

[186] Zhuangwei Zhuang et al. "Discrimination-aware Channel Pruning for Deep Neural Networks". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 881–892. URL: http://papers.nips.cc/paper/7367-discrimination-aware-channel-pruning-for-deep-neural-networks.pdf.