



VCU

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations


Graduate School

2020

Invariance and Invertibility in Deep Neural Networks

Han Zhang
Virginia Commonwealth University

Follow this and additional works at: <https://scholarscompass.vcu.edu/etd>

 Part of the [Artificial Intelligence and Robotics Commons](#), [Data Science Commons](#), [Geometry and Topology Commons](#), and the [Theory and Algorithms Commons](#)

© The Author

Downloaded from

<https://scholarscompass.vcu.edu/etd/6358>

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

© Han Zhang, May 2020

All Rights Reserved

INVARIANCE AND INVERTIBILITY IN DEEP NEURAL NETWORKS
A dissertation submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy at Virginia Commonwealth University.

by

HAN ZHANG

Master of Science in Mathematical Sciences from VCU, 2015
Bachelor of Science in Mathematical Sciences from VCU, 2013

Director: DR. TOMASZ ARODZ
ASSOCIATE PROFESSOR, DEPARTMENT OF COMPUTER SCIENCE

Virginia Commonwealth University
Richmond, Virginia
May, 2020

Contents

List of Figures	3
List of Tables	5
List of Algorithms	5
Abstract	7
1 Introduction	8
1.1 History of Neural Networks	8
1.2 Invariant Machine Learning Models	9
1.3 Invertible Machine Learning Models	9
1.4 Research Aims	9
2 Background on Neural Networks	11
2.1 Abstract View of Differentiable Models	11
2.1.1 Computation Graph	11
2.1.2 Automatic Differentiation (AD)	12
2.2 Feed-forward Neural Networks	14
2.2.1 Single-layer Perceptron	14
2.2.2 Multi-layer Perceptron	15
2.2.3 Gradient-Based Learning	17
2.2.3.1 Batch Gradient Descent	20
2.2.3.2 Stochastic Gradient Descent (SGD)	20
2.2.3.3 Momentum	22
2.2.4 Challenges in Neural Network Optimization	22
3 Learning Invariant Layers	23
3.1 Feed-forward Networks are not well-suited for Learning Invariance	25
3.2 Proposed Invariant Architecture and Its Experimental Validation	28
3.3 Learning Invariant Representations from Relational Data	31
3.4 Experimental Validation of Learning Invariant Representations from Data	32
4 Learning Invertible Neural Blocks	36
4.1 Architectures for Invertible Neural Networks	36
4.1.1 Neural ODEs	36
4.1.2 Invertible Residual Networks	37
4.1.3 Limitations of Approximation Capabilities of Neural ODEs and i-ResNets	37
4.2 Augmented Neural ODEs are Universal Approximators	38

4.3	Background on ODEs, Flows, and Embeddings	39
4.3.1	Correspondence between Flows and ODEs	39
4.3.2	Flow Embedding Problem for Homeomorphisms	39
4.4	Approximation of Homeomorphisms by Neural ODEs	40
4.4.1	Restricting the Dimensionality Limits Capabilities of Neural ODEs	40
4.4.2	Neural ODEs with Extra Dimensions are Universal Approximators for Homeomorphisms	41
4.5	Approximation of Homeomorphisms by i-ResNets	43
4.5.1	Restricting the Dimensionality Limits Capabilities of i-ResNets	43
4.5.2	i-ResNets with Extra Dimensions are Universal Approximators for Homeomorphisms	44
4.6	Experimental Results	45
4.6.1	Neural ODEs	45
4.6.2	i-ResNets	46
4.7	Conclusions	47
5	Conclusions and Future Work	48
Appendix A	Review on Topology and Manifold Theory	49
A.1	Topological Spaces	49
A.2	Bases, Countability, and Compactness	51
A.3	Subspaces, Products and Disjoint Unions	51
A.4	Quotient Spaces	52
A.5	Smooth Manifolds	53
A.5.1	Topological Manifolds	53
A.5.2	Smooth Structures	54
A.5.3	Manifolds with Boundary	56
A.5.4	Smooth Maps	57
A.6	Embeddings	58
A.6.1	Maps of Constant Rank	58
A.6.2	Smooth Embeddings	59
A.7	Submanifolds	59
A.7.1	Embedded Submanifolds	59
A.7.2	Restricting Maps to Submanifolds	61
A.8	The Whitney Theorems	61
A.8.1	The Whitney Embedding Theorem	61
A.8.2	The Whitney Approximation Theorems	61
A.9	Flows	62
Appendix B	List of Abbreviations	63
Appendix C	List of Notations	64
C.1	Notations in Chapter 2	64
C.2	Notations in Chapter 3	65
C.3	Notations in Chapter 4	65
	Bibliography	67
	Vita	71

List of Figures

2.1	Computation graph of expression $o = (I_1 + I_2) * (I_3 + I_4)$	11
2.2	The j -th neuron in the i -th layer: $a_{i,j} = g_i(w_{i,j,1} \cdot a_{i-1,1} + w_{i,j,2} \cdot a_{i-1,2} + w_{i,j,3} \cdot a_{i-1,3} + b_i)$	15
2.3	Single-layer Perceptron.	16
2.4	An Example of Multi-layer Perceptron with sample x as input.	16
2.5	Activation Functions	18
3.1	The general structure of an auto-encoder mapping inputs x to outputs y which are the reconstructed input data, through a latent representation or code z . An auto-encoder has two components: the encoder f (mapping x to z) and the decoder g (mapping z to y).	25
3.2	Total MSE of each activation function for learning Expression I of z by the five sizes network width in table 3.1. Plots on top show the results of the dataset of 10 features and 20 features on the bottom.	27
3.3	Total MSE of each activation function for learning Expression II of z by the five sizes network width in table 3.1. Plots on top show the results of the dataset of 10 features and 20 features on the bottom.	27
3.4	Total MSE for activation functions ReLU, SELU, Sigmoid, and Tanh with $\nu = 8$. First two plots: the middle layer is trained to approximate Expression I of z . Expression II of z for last two plots.	27
3.5	MSE for each part of code z with activation function SELU ($\nu = 8$). First two plots: the middle layer is trained to approximate Expression I of z . Expression II of z for last two plots.	28
3.6	MSE for training the full auto-encoder (left), just the encoder (center), and just the decoder (right) with activation function SELU and $\nu = 8$. Plots on top show the results of the dataset of 10 features and 20 features on the bottom. Red lines show the results for evaluating Expression I of z and black lines for Expression II of Z	28
3.7	$input_1$ vs $O_{1,1}$ and $input_2$ vs $O_{1,2}$	29
3.8	Plots of O_2 , O_3 , O_4 and O_5	30
3.9	Comparison of activation functions SELU and the new proposed activation function f_{mixed} for $\nu = 8$. Blue lines show the results for evaluating Expression I of z and yellow lines for Expression II of Z	30
3.10	Total MSE in f_{mixed} by five different sizes of widths of the network hidden layers with $\nu \in [0.6, 1, 2, 4, 8]$	31
3.11	MSE for reconstruction and for approximating the inverse distances for $\nu = 8$	32
3.12	MSE for reconstruction and for approximating the inverse distances for two-part samples.	33
3.13	MSE for reconstruction and for approximating the inverse distances for two-uneven-parts samples.	33
3.14	MSE for reconstruction and for approximating the inverse distances for interlaced two-part samples.	34

3.15	MSE for reconstruction and for approximating the inverse distances for randomly-shuffled columns, two-part samples.	34
3.16	MSE of reconstruction and inverse distances for invariance to scalar multiplication.	34
3.17	MSE of reconstruction and inverse distances for invariance to both shifting and scalar multiplication.	35
4.1	Proposed flow in \mathbb{R}^{2p+1} that embeds an $\mathbb{R}^p \rightarrow \mathbb{R}^p$ homeomorphism. Three examples for $p = 1$ are shown from left to right, including the mapping $h(x) = -x$ that cannot be modeled by a flow on \mathbb{R}^p , but can in \mathbb{R}^{2p+1}	41
4.2	Training set cross-entropy loss, for increasing number d of null channels, added to RGB images. For each d , the input images have dimensionality $32 \times 32 \times (3 + d)$. Left: ODE-Net with 16 convolutional filters; center: 64 filters; right: 128 filters.	46
4.3	Test set cross-entropy loss, for increasing number d of null channels, added to RGB images. For each d , the input images have dimensionality $32 \times 32 \times (3 + d)$. Left: ODE-Net with 16 convolutional filters; center: 64 filters; right: 128 filters.	46
4.4	Minimum of cross-entropy loss across all epochs as a function of d , the number of null channels added to input images, for ODE-Nets with a different number of convolutional filters; $d = 4$ corresponds to $q \geq 2p + 1$. Left: training set loss; right: test set loss.	47

List of Tables

2.1	Forward Mode of $o = (I_1 + I_2) \star (I_3 + I_4)$ evaluated at $(I_1, I_2, I_3, I_4) = (1, 2, 3, 4)$. .	13
2.2	Reverse Mode of $o = (I_1 + I_2) \star (I_3 + I_4)$ evaluated at $(I_1, I_2, I_3, I_4) = (1, 2, 3, 4)$. .	13
3.1	Widths of the neural networks used in experiments in this Chapter.	26
B.1	List of Abbreviations	63
C.1	Notations in Chapter 2: Fig. 2.1	64
C.2	Notations in Chapter 2: Sizes and Dataset	64
C.3	Notations in Chapter 2 and Chapter 3: Neural Network	65
C.4	Notations in Chapter 3	65
C.5	Notations in Chapter 4: Dataset	65
C.6	Notations in Chapter 4: Neural ODEs	66

List of Algorithms

1	Batch Gradient Descent.	21
2	Stochastic Gradient Descent (SGD).	21
3	Stochastic Gradient Descent (SGD) with momentum.	22

Abstract

INVARIANCE AND INVERTIBILITY IN DEEP NEURAL NETWORKS

by Han Zhang

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2020.

Director: Dr. Tomasz Arodz

Associate Professor, Department of Computer Science

Machine learning is concerned with computer systems that learn from data instead of being explicitly programmed to solve a particular task. One of the main approaches behind recent advances in machine learning involves neural networks with a large number of layers, often referred to as deep learning. In this dissertation, we study how to equip deep neural networks with two useful properties: invariance and invertibility. The first part of our work is focused on constructing neural networks that are invariant to certain transformations in the input, that is, some outputs of the network stay the same even if the input is altered. Furthermore, we want the network to learn the appropriate invariance from training data, instead of being explicitly constructed to achieve invariance to a pre-defined transformation type. The second part of our work is centered on two recently proposed types of deep networks: neural ordinary differential equations and invertible residual networks. These networks are invertible, that is, we can reconstruct the input from the output. However, there are some classes of functions that these networks cannot approximate. We show how to modify these two architectures to provably equip them with the capacity to approximate any smooth invertible function.

Chapter 1

Introduction

In recent years, Machine Learning (ML) has seen rapid growth due to the high amount of data produced in many industries and the increase in computation power. Machine learning allows for developing computer systems that can learn from data to perform a task without being explicitly programmed with a specific set of steps for completing that task. Machine Learning algorithms are used everywhere, from routine business analytics to complex projects in artificial intelligence like self-driving cars. We can use ML for various purposes: predictions, image recognition, language translation, medical diagnoses, and others.

1.1 History of Neural Networks

One of the main approaches to powering current ML systems is deep learning. Deep learning algorithms use the structure of the neural network to find connections between inputs and outputs to achieve a specific purpose. The reason why it is called deep learning is that the structure of deep neural network consists of a large number of layers compared to early neural networks, more weights and biases to adjust the outputs to improve the capabilities to approximate vast amounts of complex data.

Neural networks can be traced back to 1943 when McCulloch and Pitts [1] wrote a paper on how neurons might work, and tried to understand how the brain could produce highly complex patterns by using many basic cells that are connected. In 1958, Rosenblatt [2] invented the first perceptron - a single artificial neuron that was built in hardware. It is the oldest neural network unit and is still in use today. A single-layer perceptron was found to be useful in binary classification. However, perceptron was a linear unit, and it was shown that even if many layers of linear units are stacked on top of each other, the resulting network cannot move beyond linear classification [3]. This observation has significantly reduced interest in neural networks for over a decade.

In 1960, Kelley [4] showed the first-ever version of the continuous back-propagation model. In 1962, Dreyfus [5] developed a simpler version, which only relies on the chain rule. Back-propagation gained prominence in the neural network community as a method for training multi-layered networks with nonlinearities after a seminal paper by Rumelhart et al. in 1986 [6], resulting in renewed interest in neural networks. The first practical demonstration a convolutional neural network [7] with back propagation on handwritten digits has been shown soon after [8]. A method called max-pooling [9] was introduced in 1993 and it can help with shift-invariance and tolerance to

deformation to aid in 3D object recognition. After this second period of significant developments, in mid-1990s, a group of alternative, easier to train machine learning approaches emerged [10], reducing interest in neural networks again.

In the last decade, interest in neural networks revived again, fueled by rapid developments in parallel processing on GPUs, availability of large training datasets, and improvements in neural architectures, which together allowed for training much larger networks. In 2016, [11] provided a neural network method that first time ever defeated human professional player in a complex game.

High-performing neural networks often use two architectural building blocks: convolutional layers and residual layers. Convolutional neural networks (CNNs) have achieved excellent results on various visual recognition tasks [12]. While CNNs learn the same filter to be applied to different regions of the input, the output of the layer still is position-specific. Thus convolution layer alone is not translation invariant. With the addition of a pooling layer, CNNs become locally transform-invariant. However, larger changes in the positions of input features [12, 13], such as distortions or shifting can cause the positions of features to vary. Residual networks (ResNet) [14] have been proposed as a way to improve training of very deep networks. Instead of learning an output for a given input from scratch, residual networks propose that the network should learn what needs to be modified in the input to produce the output, which often is an easier task.

1.2 Invariant Machine Learning Models

Invariance to abstract input features is a highly desirable property for many tasks [15]. More abstract concepts are generally invariant to most local changes of the input [16]. Many unsupervised feature learning algorithms have emerged for learning representations from data, and it is important to learn invariant representations in high-dimensional data [17]. However, many algorithms can only learn invariant representations for specific transformations, for example convolution operators can be exploited to learn shift-invariance [18, 19, 20]. The denoising auto-encoder [21] can learn robust features of the input noise by attempting to recreate the original data from a hidden representation of the disrupted data. However, it is still a challenging problem to learning invariant representations for types of transformations that are not known in detail upfront.

1.3 Invertible Machine Learning Models

Another useful property of machine learning models is invertibility: the ability of reconstructing the input from the model's output. Typical neural networks are not invertible, some information is lost when it passes through layers from input to output. In recent years, residual networks have been modified to result in architectures that guarantee invertibility. These are finite-depth Invertible Residual Networks (i-ResNets) [22] and their infinite-depth generalization, Neural ODEs (ODE-Nets) [23]. Unfortunately, recent evidence shows that Neural ODEs are not capable of performing arbitrary data transformations, for example, it cannot learn to produce $-x$ on output if given x on input, no matter how large then ODE-Net is.

1.4 Research Aims

In this dissertation, we go beyond standard architectures of neural networks to explore two properties: invariance and invertibility. We first analyze regular feed-forward neural networks and

show that they are not effective in learning invariant mappings. Then, we propose a new architecture that is more capable at approximating invariant mappings, and can learn invariance from data. Second, we explore two invertible architectures, Neural ODEs and i-ResNets, and we show that while out-of-the-box these are not universal approximators, they can be turned into universal approximators with one simple improvement.

We organize this dissertation as follows. We first review the background on deep neural networks in Chapter 2. It describes how a neural network works and how to train a neural network. Then, in Chapter 3, we present the invariance model with a new activation function. We conduct experiments to show that it can achieve better results in learning invariance than standard neural networks. In chapter 4, we focus on the approximation capabilities of invertible neural networks. Chapter 4 requires some knowledge in topology and manifold theory, which we provide in the Appendix. Finally, Chapter 5 presents conclusions and outlines proposed further work. We also summarize the notations in Appendix C.

Chapter 2

Background on Neural Networks

Neural networks are a branch of machine learning models, and they can be used in many fields, including image recognition, speech recognition, and natural language processing, and they generally provide competitive solutions in these fields. In this section, we outline a basic understanding of the architecture of neural networks and how they work. A neural network can be seen as a model defined by specifying an abstract computational graph. The model is trained using gradient descent methods relying on automatic differentiation of the computational graph. We describe the computational graph in section 2.1.1. Then, section 2.1.2 explains automatic differentiation, a generalized back-propagation which is used to describe the evaluation of derivatives efficiently. These derivatives are then used to make adjustments to the weights to train the neural network. After introducing this abstract view that is prevalent in most current neural network libraries, we focus on specific forms of computational graph defining feed-forward neural networks - we describe essential elements of these networks, such as neurons, weights, bias, activation function, and layers, in more detail in Section 2.2. We also describe in more detail the training procedure that uses gradients to improve the networks.

2.1 Abstract View of Differentiable Models

2.1.1 Computation Graph

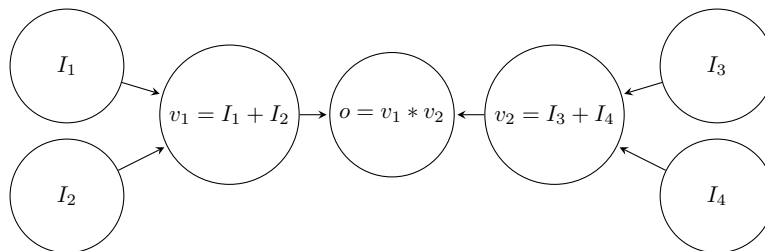


Figure 2.1: Computation graph of expression $o = (I_1 + I_2) * (I_3 + I_4)$.

A computational graph (Fig. 2.1) is a directed graph that shows a mathematical expression in

an intuitive way as a sequence of simpler expressions. Each node in a computational graph is a variable. The variable is either an input or a parameter of the network, or is a result of an operation involving other variables. Variables can be of scalar, vector, matrix, or tensor type. An operation is usually a simple function of one or more variables appearing earlier in the computational graph – that is, variables can send their values to an operation, and an operation can also take the inputs from other operations. These connections between variables are represented by the edges in the computational graph. Edges represent function arguments, and they are pointers between nodes with directions. For example, consider the expression $o = (I_1 + I_2) * (I_3 + I_4)$ with its corresponding computation graph in Fig. 2.1. This expression has four input variables and three operations: two additions and one multiplication. Two intermediate variables $v_1 = I_1 + I_2$ and $v_2 = I_3 + I_4$ are introduced, and thus, $o = v_1 * v_2$. Therefore in a computational graph, each operation and input variable becomes a node; the output of one node goes to another node with arrows as direction. A neural network can be seen as a very large computational graph, transforming inputs into outputs, and having parameters or weights that allow for changing the transformation to fit the desired outputs more closely.

2.1.2 Automatic Differentiation (AD)

Automatic differentiation (AD) applies the chain rule of differential calculus to a computational graph. Derivatives of the graph outputs with respect to some variables – those representing network weights – allow for finding better values for these variables – that is, for training the neural network. AD computes derivatives through the accumulation of intermediate values and generates numerical derivative evaluations rather than a symbolic expression for the derivative, which gives AD a two-sided nature that is partly symbolic and partly numerical. In computational graphs, all numerical computations are formed by a finite set of elementary operations with known derivation rules. We can obtain the derivative of the overall composition by combining all the derivatives of elementary operations through the chain rule. Typically, elementary operations include addition, multiplication, switching signs, reciprocal, and standard special functions such as exp, sine. We can list the evaluation trace of elementary operations, and this forms the basis of the technique of AD. A computation graph can also represent evaluation traces in visualizing dependency relations between intermediate variables [24]. AD has two modes: forward accumulation mode or simply forward mode and reverse accumulation mode or reverse mode.

AD in forward mode is very straightforward. Forward mode follows chain rule and computes partial derivatives [25] of every variable with respect to input variables – including variables corresponding to network weights – at the same time as it performs forward evaluation of every variable. Consider a simple example of the evaluation trace of the computation graph in Fig. 2.1 with $(I_1, I_2, I_3, I_4) = (1, 2, 3, 4)$. In this example, the main goal of the forward mode is to compute the derivative of o with respect to I_1 . We first pass the values of the inputs through the network (Table 2.1a). The corresponding derivatives for each variable in Table 2.1b are evaluated in lock-step with the variables in the forward primal trace by applying chain rule to obtain the desired derivative in the last variable, $\partial o / \partial I_1$.

AD in the reverse mode is a generalized back-propagation algorithm. It calculates derivatives of the output variables with respect to other variables, and propagates derivatives backward from a given output. To achieve this goal, it complements each variable with another variable called an adjoint. An adjoint for a given variable contains the partial derivative of the output output with respect to the given variable. Adjoint are calculated backwards, from the output towards

I_1	= 1
I_2	= 2
I_3	= 3
I_4	= 4
$v_1 = I_1 + I_2$	= 3
$v_2 = I_3 + I_4$	= 7
$o = v_1 * v_2$	= 21

(a) Forward Primal Trace

$\partial I_1 / \partial I_1$	= 1
$\partial I_2 / \partial I_1$	= 0
$\partial I_3 / \partial I_1$	= 0
$\partial I_4 / \partial I_1$	= 0
$\partial v_1 / \partial I_1 = \partial I_1 / \partial I_1 + \partial I_2 / \partial I_1$	= 1 + 0 = 1
$\partial v_2 / \partial I_1 = \partial I_3 / \partial I_1 + \partial I_4 / \partial I_1$	= 0 + 0 = 0
$\partial o / \partial I_1$	
$= v_2 * \partial v_1 / \partial I_1 + v_1 * \partial v_2 / \partial I_1$	= $b = 7$

(b) Forward Derivative Trace

Table 2.1: Forward Mode of $o = (I_1 + I_2) * (I_3 + I_4)$ evaluated at $(I_1, I_2, I_3, I_4) = (1, 2, 3, 4)$.

the input, and eventually the partial derivative of the output with respect to the input is obtained. Below, a variable with a bar over it is used to denote its corresponding adjoint. For example I_1 can only affect output o through affecting v_1 , so its contribution to the change in o is given by

$$\frac{\partial o}{\partial I_1} = \frac{\partial o}{\partial v_1} \cdot \frac{\partial v_1}{\partial I_1} \text{ or } \bar{I}_1 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial I_1}. \quad (2.1)$$

In eq. (2.1), o is a given output for v_1 and v_1 is given output for I_1 , and the contribution of I_1 to the change in o is computed in two steps in table 2.2

$$\bar{v}_1 = \bar{o} \cdot \frac{\partial o}{\partial v_1} \text{ and } \bar{I}_1 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial I_1} \quad (2.2)$$

Typically, in training a neural network, there is a single output equal to some measure of the current error of the network. That is, the output is a single scalar, and in the reverse mode, we need a single adjoint per variable, the partial derivative of the error with respect to that variable. The forward phase populates every intermediate variable in the computation graph and records each dependency thorough a bookkeeping procedure. Then in the reverse mode, each partial derivative is calculated by propagating adjoints from the outputs to the inputs. Table 2.2 shows the corresponding adjoint of each variable in table 2.1a. The reverse mode [24] is significantly more efficient to evaluate (in terms of operation count) than the forward mode for functions with a large number of inputs.

$\bar{I}_1 = \bar{v}_1 \cdot \partial v_1 / \partial I_1$	= $\bar{v}_1 \times 1$	= 7
$\bar{I}_2 = \bar{v}_1 \cdot \partial v_2 / \partial I_2$	= $\bar{v}_1 \times 1$	= 7
$\bar{I}_3 = \bar{v}_2 \cdot \partial v_2 / \partial I_3$	= $\bar{v}_2 \times 1$	= 3
$\bar{I}_4 = \bar{v}_2 \cdot \partial v_2 / \partial I_4$	= $\bar{v}_2 \times 1$	= 3
$\bar{v}_1 = \bar{o} \cdot \partial o / \partial v_1$	= $\bar{o} \times v_2$	= 7
$\bar{v}_2 = \bar{o} \cdot \partial o / \partial v_2$	= $\bar{o} \times v_1$	= 3
$\bar{o} = \partial o / \partial o$	= 1	

Table 2.2: Reverse Mode of $o = (I_1 + I_2) * (I_3 + I_4)$ evaluated at $(I_1, I_2, I_3, I_4) = (1, 2, 3, 4)$.

2.2 Feed-forward Neural Networks

A neural network is a form of a computational graph. Neurons, also called nodes or units, are fundamental units in a neural network. A neuron works the same as the nodes in a computational graph, and it can receive inputs from some other neurons or external input and form some operations on its inputs. In the computation graph, the edges only point the direction between two nodes. In a neural network, each edge carries another variable called a connection weight, which has a magnitude assigned to it. The weights stand for the relative importance of the connection from the edge source neuron to edge target neuron. Within the neuron, the weighted sum of neuron input values is calculated, and then an activation function is applied. In the bigger picture, we organize neural networks in layers and aggregate several or many layers into one neural network. Usually, one layer is composed of many neurons.

Fig. 2.2 shows an example of a single neuron (the j -th neuron in the i -th layer) that takes 3 inputs $a_{i-1,1}$, $a_{i-1,2}$ and $a_{i-1,3}$, which are associated with weights $w_{i,j,1}$, $w_{i,j,2}$ and $w_{i,j,3}$ respectively. Additionally, there is another type of factor affecting the neuron's output, $b_{i,j}$, called a bias. The weighted sum of its inputs is then calculated with an activation function g_i applied to the sum. In this section, we only consider the case when the input of a neuron is a vector; for example, the x in Fig. 2.2. Neurons in different layers perform different types of transformation (activation function) on their inputs from their preceding layer. For example, let's suppose the computational graph in Fig. 2.1 is a simple neural network. The four input nodes I_1 , I_2 , I_3 , and I_4 form the input layer, and the single neuron in the middle is the output layer. The two nodes of intermediate variables form one single hidden layer, which transforms inputs into v_1 and v_2 . Then the output layer transforms the hidden layer into the final output o .

There are many types of neural networks with different architectures, for example, feed-forward neural network, convolutional neural network (CNN), recurrent neural network (RNN), and recursive neural network. The architecture of a feed-forward network is the simplest and most fundamental among other types of neural networks, and only feed-forward neural network is used in this dissertation. As mentioned before, some connected layers form a neural network, and each layer is a group of neurons. Neurons in the same layer take the outputs from their preceding layer or external source individually and work at the same time. In a feed-forward neural network, the connections between neurons in different layers do not form a cycle or a loop, and the outputs of each layer only move in the direction from the input layer through hidden layers (if there are any) and then to the output layer. When counting the depth of a deep neural network, only the layers that have tunable weights are considered. There are two types of feed-forward neural networks, single-layer perceptron and multi-layer perceptron (MLP).

2.2.1 Single-layer Perceptron

Single-layer perceptron only has one neuron and, simply speaking, it is a neuron with threshold function as the activation function. It is typically used as a binary classifier that uses the threshold function as activation function (Fig. 2.3a): a function that takes an example x as input and maps it to an output value $g_1(x)$. The output $g_1(x)$ first calculates the weighted sum of all its inputs and then adds the bias term b_1 , $\sum_{i=1}^{i=n} w_{1,1,i} \cdot x_i$ before applying the activation function. The activation function g_1 takes the comparison of the weighted sum with 0, if it's larger than 0, the neuron 'fires'

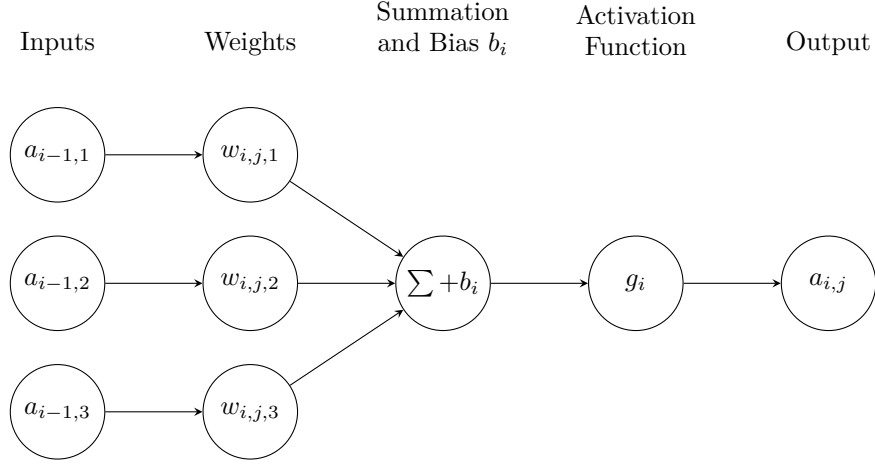


Figure 2.2: The j -th neuron in the i -th layer: $a_{i,j} = g_i(w_{i,j,1} \cdot a_{i-1,1} + w_{i,j,2} \cdot a_{i-1,2} + w_{i,j,3} \cdot a_{i-1,3} + b_i)$.

(output $g_1(x) = 1$) otherwise it doesn't 'fire' (output $g_1(x) = 0$).

$$\hat{y} = g_1(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{i=n} w_{1,1,i} \cdot x_i > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

The function $g_1(x)$ in eq. (2.3) can decide whether or not an input sample belongs to some specific class. The single-layer perceptron is a type of linear classifier: we can separate the data by a linear decision boundary, and the bias term b_1 allows the classifier $g_1(x)$ to shift the decision boundary within the input space. This binary case can also be extended to predict any number of classes with multiple single-layer perceptrons if each class is linearly separable from other classes. For example, there are c classes to be classified. Each class can be separated from other $c - 1$ classes by one decision boundary; thus, these c classes can be separated by $c - 1$ decision boundaries or single-layer perceptrons with $c - 1$ neurons in the output layer.

2.2.2 Multi-layer Perceptron

Multi-layer perceptron network (MLP) consists of at least three layers. Besides the input and output layers we have seen in the single-layer perceptron, there are also one or more hidden layers.

Fig. 2.4 shows an example of MLP, and it consists of an input and an output layer with two hidden layers. In most cases of deep learning, a neural network consists of at least one hidden layer, so the term of MLP sometimes stands for feed-forward neural network in general. Typically, neurons in the same layer use the same type of activation function. The activation functions in the hidden layers must be nonlinear. Connecting more than one linear layer in a sequence is not useful, as this is equivalent to just one single linear layer. As MLP contains multiple layers, and each layer introduces non-linearities, MLP can solve linearly non-separable problems. For simplicity (in the rest of this section), for a random neuron from MLP: u is used to denote the weighted sum of elements in the input vector, and g is the corresponding activation function with the neuron's output $a_{i,j}$ as $g(u)$. The rest of this section shows some commonly used activation functions.

If we do not have a linear neuron, that is, no activation function g , a neuron outputs u directly, and the output of this neuron is not confined within any range (Fig. 2.5a, green line). In most

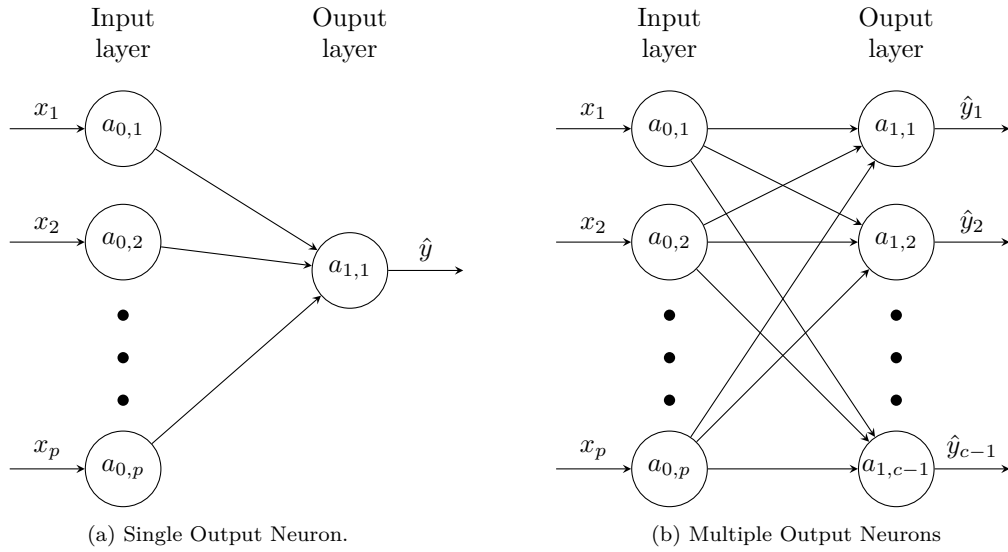


Figure 2.3: Single-layer Perceptron.

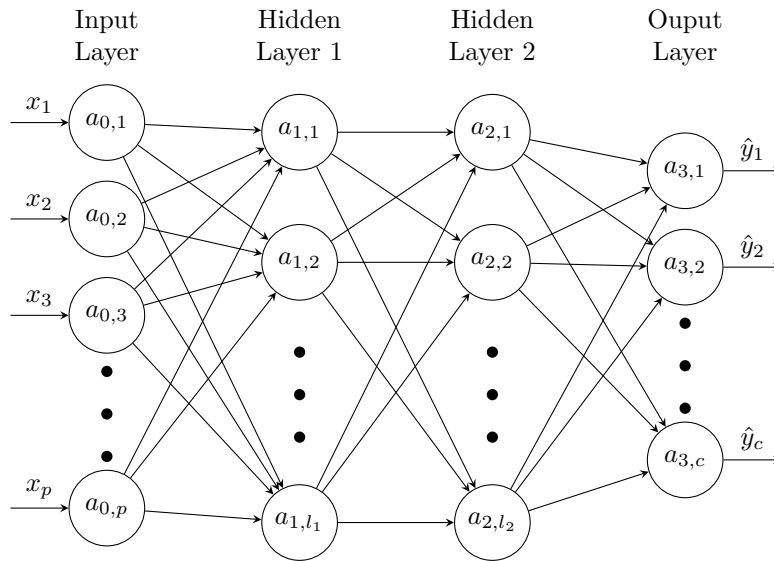


Figure 2.4: An Example of Multi-layer Perceptron with sample x as input.

cases, we are interested in possibly nonlinear transformation of complex input data, and to achieve that, we need nonlinear activation functions. We review several most popular activation functions below.

The sigmoid activation function (Fig. 2.5a, blue line) is in the form of,

$$g(u) = \frac{1}{1 + e^{-u}} \quad (2.4)$$

and it squeezes the values of u to the range 0 and 1. As probabilities are between 0 and 1, the output of sigmoid can be interpreted as probabilities.

The bipolar sigmoid (tanh) activation function is defined as

$$g(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}} = \frac{1 - e^{-2u}}{1 + e^{-2u}} = \frac{2 - (1 + e^{-2u})}{1 + e^{-2u}} = \frac{2}{1 + e^{-2u}} - 1 \quad (2.5)$$

The $\frac{1}{1+e^{-2u}}$ in Eq. (2.5) is a sigmoid function with range 0 and 1, the tanh activation function (Fig. 2.5a, red line) is a rescaled and shifted sigmoid function. Thus, tanh maps the inputs to the range -1 and 1. Tanh maps zero values to 0, and this is different from the sigmoid function, which centers at 0.5. We can use tanh in hidden layers, but also for binary classification.

ReLU activation function is defined as:

$$g(u) = \max(0, u). \quad (2.6)$$

It is the most widely used default activation function in recent years.

ReLU (Fig. 2.5b, red line) pushes negative inputs to be zero, and equals the inputs itself when inputs are positive. When using ReLU as an activation function, the convergence speed is much faster than sigmoid or tanh, because gradients are always constant for positive outputs; also, when we have negative inputs, ReLU outputs them as zero. It could also decrease the ability to model the inputs accurately when using ReLU, and many other variants of ReLU are introduced to solve this problem. For example, Leaky ReLU, instead of outputting 0 when the inputs are non-positive, the output is 0.01 times the input ($g(u) = \max(0.01u, u)$, Fig. 2.5c), which gives some flexibility of modeling the inputs, but the range of the output is changed to $-\infty$ and ∞ . Soft-plus activation function, which has the following equation, is often used as well:

$$g(u) = \ln(1 + e^u) \quad (2.7)$$

Soft-plus (Fig. 2.5b, blue line) is similar to ReLU but smoother and differentiable around 0. But in contrast to ReLU, as a soft-plus function contains operations of the exponential and natural logarithm which are more difficult to be computed than ReLU when it comes to derivatives. SELU (Fig. 2.5d) has two fixed parameters $\alpha = 1.6732$ and $\lambda = 1.0575$, and it is similar to ReLU for values larger than zero. Compared to ReLU, SELU not only has no problem with vanishing gradients but also tends to increase the speed of learning.

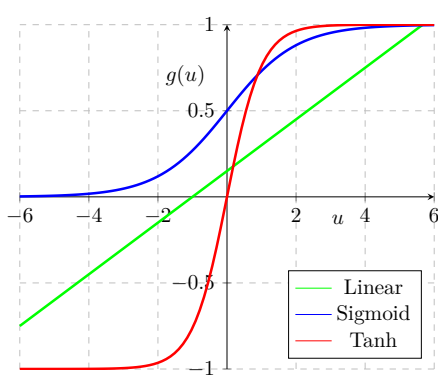
$$a = \lambda \begin{cases} u & \text{if } u > 0 \\ \alpha e^u - \alpha & \text{if } u \leq 0 \end{cases} \quad (2.8)$$

2.2.3 Gradient-Based Learning

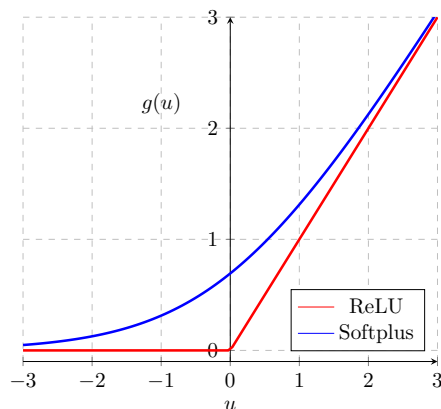
We know that neural networks are organized by a group of layers in a chain structure, and each layer is composed of some neurons. Each layer performs some operations on the outputs from the other layer. Thus each layer is a function of its preceding layer. In this structure, if the input layer is $a_0 = x$, then the general activation formula for a neural network with L layers is

$$a_l = g_l(w_l \cdot a_{l-1} + b_l), \quad l \in \{1, 2, \dots, L\}. \quad (2.9)$$

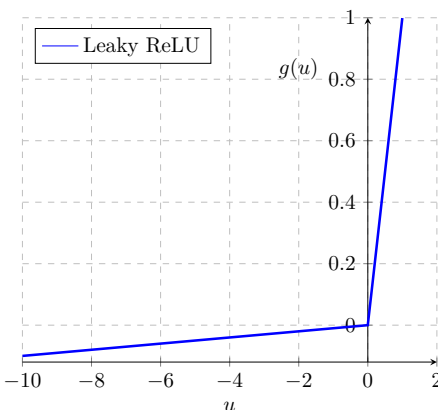
To design a neural network, we need to choose the number of layers, number of neurons in each layer, and activation functions. Neural networks can be trained with gradient descent, just like



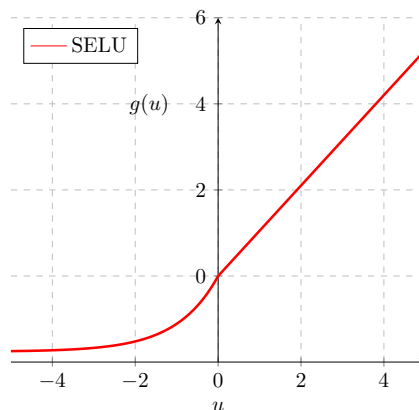
(a) Linear, Sigmoid and Tanh



(b) ReLU and Soft-plus



(c) Leaky ReLU. When $u > 0$, Leaky ReLU has the same outputs as ReLU. When $u < 0$, Leaky ReLU outputs $0.01 * u$ instead of 0.



(d) SELU with $\alpha = 1.6732$ and $\lambda = 1.0507$. When $u > 0$, SELU outputs $1.0507 * u$, which is roughly the same as ReLU. When $u < 0$, SELU has an exponential shape.

Figure 2.5: Activation Functions

other machine learning algorithms. Training a machine learning algorithm requires an optimization procedure, a cost function, and the model to be used. Similarly, when training a feed-forward neural network, there also need a cost function and an optimization procedure. There are multiple choices of cost functions when training a neural network, for example, mean squared error (MSE), cross-entropy cost, and Kullback–Leibler divergence. Cost functions are often used in classification to monitor errors in predictions. Minimizing the cost function means finding the lowest possible error value; but in practice, we may only be able to find the local minimum of the cost function. A class of algorithm called gradient descent or some of its variants are used to minimize the cost function. In recent years, in practical implementations, gradient descent relies on automatic differentiation that we introduced in section 2.1.2 to efficiently calculate gradients. Below, we show why gradients are useful in minimizing the cost function by iteratively updating weights of neural network parameters.

Suppose we need to minimize a cost function $J(\theta)$ for neural network f with training dataset

consisting of m input vectors $\{x^{(1)}, \dots, x^{(m)}\}$ and m corresponding targets $\{y^{(1)}, \dots, y^{(m)}\}$

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m C(f(x^{(i)}; \theta), y^{(i)}) = \frac{1}{m} \sum_{i=1}^m C(\hat{y}^{(i)}, y^{(i)}) \quad (2.10)$$

by choosing optimal parameters θ (usually the parameters are weights of all connection between neurons). Function $C(f(x^{(i)}; \theta), y^{(i)})$ calculates the per-example cost related to the difference between the true target y and the predicted value \hat{y} . Then, cost function $J(\theta)$ is an average cost over the training set. Intuitively, minimizing the value of a cost function $J(\theta)$ means finding the bottom of the deepest valley of this cost function. The derivative can help us walk down to the bottom of a valley, though not necessarily the deepest valley.

A network has many weights; thus, $J(\theta)$ is usually a multivariate function. It is easy to begin with a one-dimensional case, a single-variable function $J(\theta)$, where the parameter θ is a single number. The derivative of cost function $J(\theta)$ at θ is defined as

$$\frac{dJ(\theta)}{d\theta} = \nabla_{\theta} J(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon} \approx \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon} \quad (2.11)$$

for some small real number ϵ . Then,

$$J(\theta + \epsilon) \approx J(\theta) + \epsilon \nabla_{\theta} J(\theta) \quad (2.12)$$

It tells us that if we change the input by a small enough number, we can obtain the corresponding change approximately in the output. We can thus use this information to minimize the cost function. Eq. 2.12 shows that $J(\theta + \epsilon)$ is smaller than $J(\theta)$ if $\epsilon \nabla_{\theta} J(\theta)$ is negative (recall that, in uni-variate case, $\nabla_{\theta} J(\theta)$ is a single number). We can thus minimize $J(\theta)$ by moving θ in small step λ in the direction given by the opposite sign of the derivative:

$$\theta = \theta - \lambda \nabla_{\theta} J(\theta) \quad (2.13)$$

The change in θ is

$$-\lambda \nabla_{\theta} J(\theta), \quad (2.14)$$

and its magnitude is controlled by λ , a hyperparameter called the learning rate.

Eq. (2.13) gives us the main idea of gradient descent. When θ has more than one variable, partial derivative is used instead of the derivative. The partial derivative of a function is its derivative with respect to a specific single variable while other variables are held constant. If there are n variables in θ (usually $n > 1$), then cost function $J(\theta)$ is a multivariate function with

$$\theta = \langle \theta_1, \theta_2, \dots, \theta_j, \dots, \theta_n \rangle \quad (2.15)$$

The partial derivative of cost function $J(\theta)$ with respect to θ_j can be written as

$$\nabla_{\theta_j} J(\theta) = \frac{\partial J}{\partial \theta_j} = \lim_{\epsilon \rightarrow 0} \frac{J(\theta_1, \theta_2, \dots, \theta_j + \epsilon, \dots, \theta_n) - J(\theta_1, \theta_2, \dots, \theta_j, \dots, \theta_n)}{\epsilon} \quad (2.16)$$

where ∂ stands for the partial derivative. When we have more than one variable in a function, the number of possible directions to reduce the cost function is infinite. To deal with this problem, we define a unit vector $v = \{v^1, v^2, \dots, v^j, \dots, v^n\}$ and calculate the rate of change in the direction of

v at a specific point, and this is called the directional derivative in the direction given by the unit vector v . With the cost function $J(\theta)$, the directional derivative is defined as

$$D_v J(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta_1 + \epsilon v^1, \theta_2 + \epsilon v^2, \dots, \theta_j + \epsilon v^j, \dots, \theta_n + \epsilon v^n) - J(\theta_1, \theta_2, \dots, \theta_j, \dots, \theta_n)}{\epsilon}. \quad (2.17)$$

The partial derivative with respect to θ_j in eq. (2.16) can be seen as a special case of directional derivative (eq. (2.17)) with $v^j = 1$ and all the other elements are all zeros, $v = \{0, 0, \dots, v^j = 1, \dots, 0\}$. Then, the gradient (the vector of all partial derivatives) of $J(\theta)$ with respect to θ is

$$\nabla_{\theta} J(\theta) = \left\langle \frac{\partial J(\theta)}{\partial \theta_1}, \frac{\partial J(\theta)}{\partial \theta_2}, \dots, \frac{\partial J(\theta)}{\partial \theta_j}, \dots, \frac{\partial J(\theta)}{\partial \theta_n} \right\rangle \quad (2.18)$$

We can express the directional derivative $D_v J(\theta)$ as a dot product of $\nabla_{\theta} J(\theta)$ and unit vector v and then we can re-write the dot product as

$$D_v J(\theta) = \nabla_{\theta} J(\theta) \cdot v = \|\nabla_{\theta} J(\theta)\| \|v\| \cos(\alpha) \quad (2.19)$$

where α is the angle between two vectors and $\| \cdot \|$ is the norm of a vector. As v is unit vector, $\|v\| = 1$. Therefore we only care about the direction of v with respect to the vector of partial derivatives. When v points in the opposite direction of $\nabla_{\theta} J(\theta)$, where $\alpha = 180^\circ$, it is the optimal direction to decrease the cost function. Thus, the equation for updating θ for the multi-variable is the same as eq. (2.13) – it involves negated gradient – but everything is in a vector form.

There are two main variants of gradient descents, batch gradient descent and stochastic gradient descent (SGD). These two variants differ in how much training data we use to compute the gradient of the cost function.

2.2.3.1 Batch Gradient Descent

In batch gradient descent, all of the m training data points are used in every iteration, and we calculate the gradient of cost $\nabla_{\theta} C(f(x^{(i)}; \theta), y^{(i)})$ at each training example i . The average changes in the cost $\sum_{i=1}^m \nabla_{\theta} C(f(x^{(i)}; \theta), y^{(i)})$ is computed by adding all the gradients at each data point i together and then divided by the number of training data points m .

$$\hat{g} = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} C(f(x^{(i)}; \theta), y^{(i)}) = \frac{1}{m} \nabla_{\theta} \sum_{i=1}^m C(f(x^{(i)}; \theta), y^{(i)}). \quad (2.20)$$

Then we use gradient descent to update the parameters θ in each iteration until the cost function $J(\theta)$ reaches a local minimum. In practice, it is often helpful to decrease the learning rate gradually over iterations, and use a learning rate at iteration k as λ_k . For batch gradient descent [26] (Algorithm 1), we need the whole training set available in memory to compute one gradient estimate, and this could cause a problem if we have a huge dataset. Usually, it needs fewer updates to the model, which is more computationally efficient than SGD, which is discussed in section 2.2.3.2. Batch gradient descent tends to be more stable in convergence, but this convergence of the model could result in a less optimal set of parameters.

2.2.3.2 Stochastic Gradient Descent (SGD)

SGD [26] in Algorithm 2 uses a strategy called mini-batch, which is a subset of the training set. For each iteration of updating the weights parameters, SGD randomly samples a mini-batch from the

Algorithm 1 Batch Gradient Descent.

Require:

Initialize learning rate λ_0 and weight parameters θ .

Training set $X = \{x^{(1)}, \dots, x^{(n)}\}$ with n examples with corresponding targets $\{y^{(1)}, \dots, y^{(n)}\}$

- 1: **while** Cost function not reaches a local minimum **do**
 - 2: Compute gradient estimate: $\hat{g} \leftarrow +\frac{1}{m}\nabla_{\theta} \sum_{i=1}^m C(f(x^{(i)}; \theta), y^{(i)})$
 - 3: Apply update: $\theta \leftarrow \theta - \lambda_k \hat{g}$
 - 4: $\lambda_{k+1} \leftarrow \text{decrease } \lambda_k$
 - 5: **end while**
-

training set and then calculate the average gradient over the mini-batch as the gradient estimate. When the size of the mini-batch is 1, this is an on-line machine learning method as we update the gradients with only one example of the training set in each iteration. On-line learning increases the frequency of updating the weight parameters θ , and this can result in faster learning on some problems, but frequent updates can introduce noisy updates. On-line learning can help the model avoid local mini-ma that is not good enough, but it may also cause the outputs of a cost function to vary and make it hard to settle on a local minimum of the cost function. The goal of SGD is to overcome the disadvantages of on-line learning and batch gradient descent, and it is the most common implementation of gradient descent used in neural networks. The size of mini-batch in practice is usually small, between 10 and 500. Thus, unlike batch gradient descent, a mini-batch can fit in the memory without any problems, and the whole training set is not needed to be stored in the memory. Like on-line learning, the frequency of updating the weight parameters in SGD is higher than batch gradient descent, and this helps avoid the early convergence that happens in the batch gradient. However, SGD still introduces some noises when updating weights parameters. The mini-batch updates provide a computationally more efficient way than on-line learning as it's less frequent to update weights in SGD. SGD requires an additional pre-chosen hyper-parameter, the size of mini-batch.

Algorithm 2 Stochastic Gradient Descent (SGD).

Require:

Initialize learning rate λ_0 , weight parameters θ , and size of mini-batch m .

Training set $X = \{x^{(1)}, \dots, x^{(m)}\}$ with m examples and corresponding targets $\{y^{(1)}, \dots, y^{(m)}\}$

- 1: **while** Cost function not reaches a local minimum **do**
 - 2: Randomly sample a mini-batch of m' samples from training set: $\{x^{(1)}, \dots, x^{(m')}\}$ with
 - 3: corresponding targets $\{y^{(1)}, \dots, y^{(m')}\}$
 - 4: Compute gradient estimate: $\hat{g} \leftarrow +\frac{1}{m'}\nabla_{\theta} \sum_{i=1}^{m'} C(f(x^{(i)}; \theta), y^{(i)})$
 - 5: Apply update: $\theta \leftarrow \theta - \lambda_k \hat{g}$
 - 6: $\lambda_{k+1} \leftarrow \text{decrease } \lambda_k$
 - 7: **end while**
-

When training a neural network, we are assuming that we have the exact gradients. But with SGD we typically have only a noisy or biased estimate of these quantities. Nonetheless, with many iterations, the noise in the gradient estimates is being reduced, while the direction of changing the parameters is maintained.

2.2.3.3 Momentum

As learning a model with gradient descent can sometimes be slow, the method of momentum [26] (Algorithm 3) is designed to accelerate the learning time. The momentum algorithm introduces a velocity vector v , that stands for the direction and speed of the parameters moving through parameter space. A hyper-parameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients decay.

Algorithm 3 Stochastic Gradient Descent (SGD) with momentum.

Require:

Initialize learning rate λ_0 , weight parameters θ , momentum parameter $\alpha < 1$, and initial velocity $v = 0$.

Training set: $X = \{x^{(1)}, \dots, x^{(n)}\}$ n examples with corresponding targets $\{y^{(1)}, \dots, y^{(n)}\}$.

- 1: **while** Cost function does not reaches a local minimum **do**
 - 2: Randomly sample a mini-batch of m' samples from training set: $\{x^{(1)}, \dots, x^{(m')}$ with
 - 3: corresponding targets $\{y^{(1)}, \dots, y^{(m')}$
 - 4: Compute gradient estimate: $\hat{g} \leftarrow +\frac{1}{m'} \nabla_{\theta} \sum_{i=1}^{m'} C(f(x^{(i)}; \theta), y^{(i)})$
 - 5: Compute velocity update: $v \leftarrow \alpha v - \lambda_k \hat{g}$
 - 6: Apply update: $\theta \leftarrow \theta + v$
 - 7: $\lambda_{k+1} \leftarrow$ decrease λ_k
 - 8: **end while**
-

2.2.4 Challenges in Neural Network Optimization

For a convex function, when we find a local minimum, it is guaranteed to be a global minimum of the function. A neural network is typically a non-convex function with many possible local minima. When training a neural network without knowing any prior information, the gradient descent likely converges to a local minimum, not the global minimum. If this local minima has a high cost compared to the global minimum, this would be a problem. This problem remains open, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost value, so it is not absolutely crucial to find the global minimum [26].

Another problem in training neural networks is the vanishing gradient problem. When we have a sigmoid activation function, it causes gradient descent trapped in flat regions, and this is because when the inputs have large absolute values, the shape of the sigmoid function tends to be flat; thus, we have small gradients. When training an MLP with many hidden layers, we need to use multiply step size with the gradients, and the product goes to zero very quickly. Therefore the weights are trapped in a relatively flat area. Because of this, the sigmoid function is rarely used in hidden layers nowadays, and ReLU is used instead. A neural network with many layers also has extremely steep regions resembling cliffs. It can be dangerous when we approach it during training, and this can be avoided by using the gradient clipping. Both of these problems can be exacerbated by long sequence of multiplications resulting from having networks with many layers.

Chapter 3

Learning Invariant Layers

It has been long recognized that data presented on the input to machine learning models can be distorted, but still represent the same object. This observation has to lead to the introduction of invariant machine learning methods, for example, techniques that ignore shifts, rotations, or light and pose changes in images. Invariant machine learning models consider inputs that differ in some way as essentially the same. For example, if we have a shifted object in an image with the same white background, we want the model to consider it the same object. In technical terms, we want the output of the model to remain constant as the input varies somehow, e.g., is shifted.

Existing approaches for invariant machine learning typically utilize pre-defined invariant features or invariant kernels and require the developer to analyze which invariance is expected in input data. In this section, we focus on invariance in deep neural networks to learn invariance from data without requiring a priori specification of the invariance type. We construct invariant layers and apply them in auto-encoder architecture, with the use of a new, proposed activation function. We show that the newly proposed activation function is more effective at handling various types of invariance than standard neural networks.

Invariance to specific types of transformations, such as image translation or rotation, has a long history in pattern recognition. Approaches based on Fourier transform [27], Zernike moments [28], and Radon transform [29], have been used to generate invariant features that can be used in downstream learning methods. Invariant kernels have also been proposed [30]. In constructing deep learning models, convolutional and pooling layers [13] offer limited amount of invariance to translation [31], and dedicated methods for achieving more robust translational invariance [32, 12], or rotation invariance [33] have been proposed. In all these cases, the type of invariance is inherent in the construction, and is fixed.

Here, we propose a new architecture for learning invariant neural networks from data. Instead of predefined which transformations of input the network should ignore, we consider a scenario where the training set consists not only of input samples but also includes information which samples are the same, up to some transformation. For example, the training set may consist of several horizontal and the vertical mirror version of an image, and based on the information that these images are essentially the same, the network should learn to become invariant to horizontal and vertical axial symmetry.

To achieve this goal, we propose a multi-layer neural network in which the output of some neurons in one of the intermediate layers to remain constant as the input varies in some way. To

train the network, we consider an auto-encoder architecture, in which the latent layer resulting from an encoder network can be partitioned into two parts: the invariant part, and the variance part.

One example of such a property, outside of neural networks, is the discrete Fourier transform. It transforms input data into the frequency domain, where the modulus is invariant to a circular shift in the input, and the phase is not invariant. Then, the inverse Fourier transform can put back the modulus and phase to reconstruct the original input. Similarly, we will use an auto-encoder architecture that reconstructs the input on the output, and in one of the intermediate layers consists of two parts, one invariant and one not.

An auto-encoder is a neural network that attempts to copy its inputs x to its outputs y , that is $y \approx x$. Internally, it has a latent intermediate layer z that describes a code used to represent some aspect of the input. Since the network's output y is supposed to be similar to input x , we typically are most interested in z , the intermediate layer, not in y . An auto-encoder is usually consisted of two parts: an encoder f that transforms the inputs x to intermediate code z ($z = f(x)$) and a decoder g that produces a reconstruction of inputs from the intermediate code $y = g(z) = g(f(x))$. Function f and g are represented by multi-layer subnetworks, the encoder and decoder, respectively. As we want the outputs y to be the reconstructed inputs x , we want the outputs y to be as close to the inputs x as possible. To achieve that, we use mean-squared error of the reconstruction as the loss

$$L(x, y) = L(x, g(f(x))) = \|x - g(f(x))\|_2^2. \quad (3.1)$$

The idea of auto-encoders can be applied to dimensionality reduction, feature learning, and pre-training of a deep neural network. In all these cases, the useful part is the encoder, which is trained to produce a low-dimensional, feature-rich representation of the input. The decoder is added to ascertain that all the information from the input is represented in the result of the encoder, z . Fig. 3.1 shows the general structure of an auto-encoder.

The intermediate code $z = f(x)$ is trained to learn representations with desired properties, for example, to be invariant. To enforce invariance, we add extra training criterion which involves comparing the outcome from the encoder z with the desired invariant representation z^* using mean-squared error (we assume for now somewhat unrealistically that z^* for each input x is known; we will eliminate this requirement later)

$$\Omega(z, z^*) = \|z - z^*\|_2^2. \quad (3.2)$$

The total objective function minimized during training the auto-encoder is then

$$C(x, y, z, z^*, a_1, a_2) = a_1 L(x, y) + a_2 \Omega(z, z^*) = a_1 \|x - g(f(x))\|_2^2 + a_2 \|f(x) - z^*\|_2^2 \quad (3.3)$$

where a_1 and a_2 are constant coefficients, we use $a_1 = 1$, $a_2 = 8$ in the experiments, indicating that we focus more on the intermediate layer than on the auto-encoder reconstruction error.

A fully-connected feed-forward neural network is capable of modeling arbitrary well-behaved functions. Thus, in principle, it can train an encoder-decoder pair that provides invariance in the intermediate code $z = f(x)$, although the network may need to be wide. Thus, our first experiment tests if a feed-forward network of width similar to the input dimensionality is capable of learning an invariant mapping. Specifically, can it learn to produce the same z^* if x are varied in some way, e.g. shifted? As an example of such a mapping, we use the modulus of Fourier transform, which is known to be invariant to translation, that is, spatial shifts of the input.

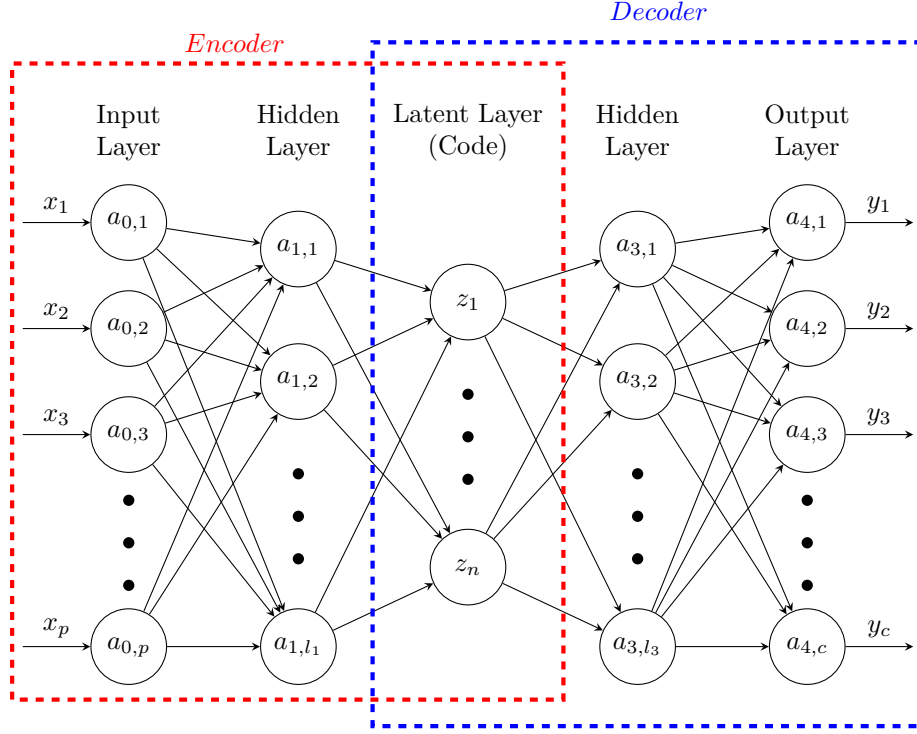


Figure 3.1: The general structure of an auto-encoder mapping inputs x to outputs y which are the reconstructed input data, through a latent representation or code z . An auto-encoder has two components: the encoder f (mapping x to z) and the decoder g (mapping z to y).

3.1 Feed-forward Networks are not well-suited for Learning Invariance

In our initial experiments, the network is supposed to learn the invariant mapping in a supervised way; that is, the dataset consists of pairs (x, z) , where $z = f(x)$ is invariant to some translation. We constructed two datasets, one with 10 features, and one with 20 features. All feature values for all samples are sampled independently from a uniform univariate distribution on $[-1, 1]$. The corresponding values of z are calculated by performing discrete Fourier transform on each sample, $z = FFT(x)$, which results in a vector of complex numbers of the same dimensionality as the input x . Then, we define four quantities, each being a real-valued vector of the same dimensionality as the input vector x .

$$\begin{aligned}
 modulus &= |z| & phase &= \text{Phase}(z) \\
 cosine &= \cos(phase) & sine &= \sin(phase)
 \end{aligned}
 \tag{3.4}$$

Of these four-vectors, the modulus is known to be invariant to input translation. Thus, there are two expressions of z which were evaluated in the experiments, they are

$$\begin{aligned}
 \text{Expression I denotes } z &= [modulus, phase] \\
 \text{Expression II denotes } z &= [modulus, cosine, sine]
 \end{aligned}
 \tag{3.5}$$

In each experiment, the encoders' output z is compared to the desired output z . The discrepancy mean-square error is used as the loss that should be minimized during training. The training set consists of 60,000 samples and features are randomly sampled independently from a uniform univariate distribution on $[-1; 1]$. We also generated a test set of 10,000 samples following the same protocol. We used stochastic gradient descent (SGD) with batch size 128 to minimize the cost function.

Four accessible nonlinear activation functions (ReLU, SELU, Sigmoid and Tanh) were tested. The experiments were tested with number of network layers in the range of 2 and 20. Five different sizes of widths of the network hidden layers are shown in Table 3.1. The formula for computing hidden layers dimensionality is

$$\text{Hidden layers dimensionality} = \text{input dimensionality} \times \text{multiplier } \nu \quad (3.6)$$

input dimensionality	multiplier ν	hidden layers dimensionality
10	0.6	6
	1	10
	2	20
	4	40
	8	80
20	0.6	12
	1	20
	2	40
	4	80
	8	160

Table 3.1: Widths of the neural networks used in experiments in this Chapter.

Fig. 3.2 and 3.3 show total MSE of each activation function for learning two expressions of z by all five sizes network width. Larger width can help the network lower the total MSE but larger number of layers cannot help lower MSE.

The result in Fig. 3.4d shows that activation function SELU is better than the other three, but none of the four activation functions leads to low MSE. We also see that sine and cosine are more accessible to be learned than a phase – the results in Fig. 3.4d considered the total MSE of both reconstruction and the invariant mapping, but the plots in Fig. 3.5 look at each component separately.

To exclude the scenario where the joint task of learning the reconstruction and the invariance makes the problem challenging, we compared the MSE for the full auto-encoder with the result for only training the encoder (i.e., learning $z = f(x)$), and for only training the decoder (i.e., learning $x = g(z)$) with activation function SELU and $\nu = 8$. Figure 3.6 shows the problem of learning with full auto-encoders comes mostly from training only the encoder as it can not learn invariant representations properly.

Taken together, the results of experiments in this section show that standard architectural building blocks are not suited well to learn invariant representations.

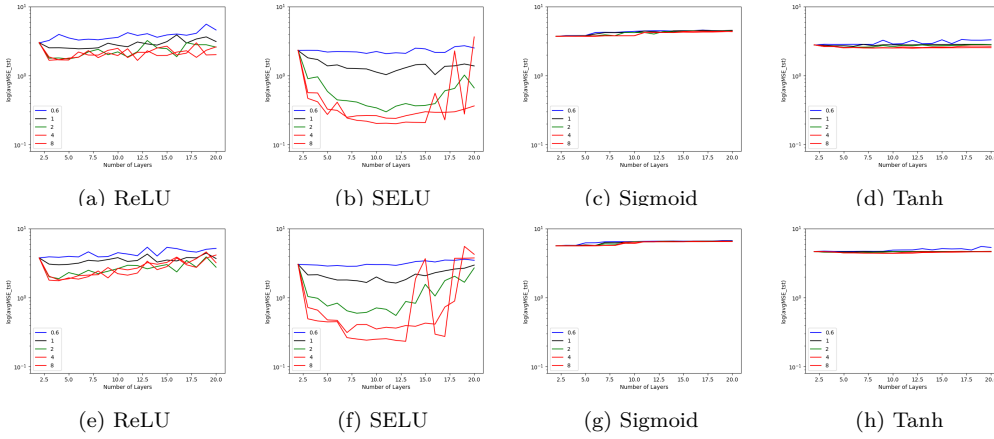


Figure 3.2: Total MSE of each activation function for learning Expression I of z by the five sizes network width in table 3.1. Plots on top show the results of the dataset of 10 features and 20 features on the bottom.

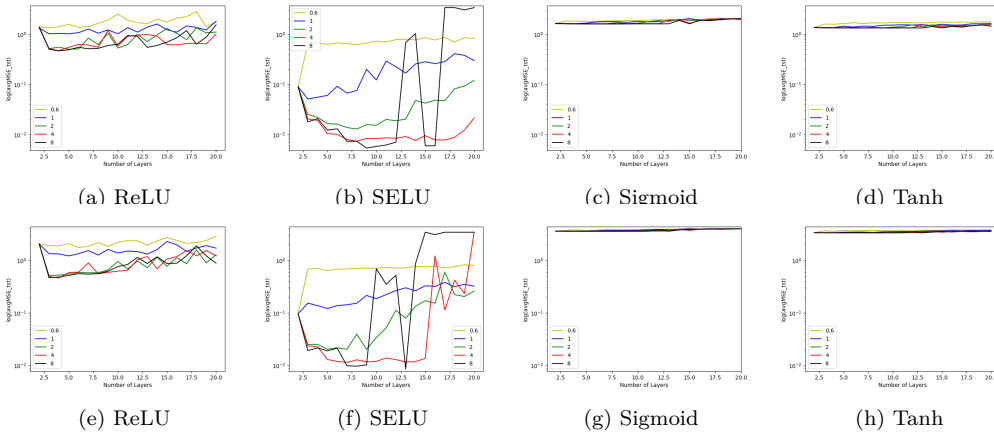
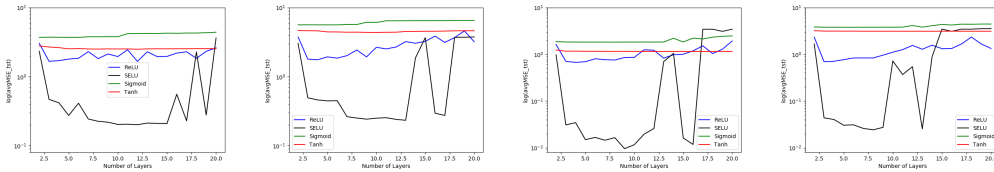
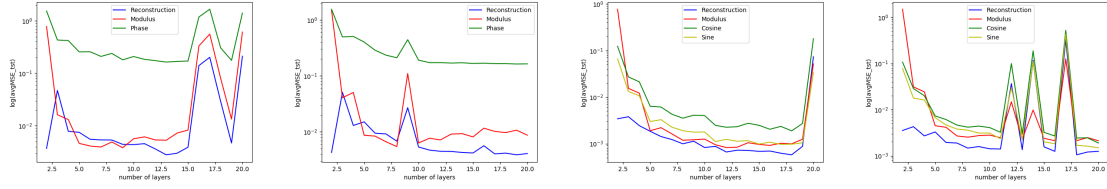


Figure 3.3: Total MSE of each activation function for learning Expression II of z by the five sizes network width in table 3.1. Plots on top show the results of the dataset of 10 features and 20 features on the bottom.

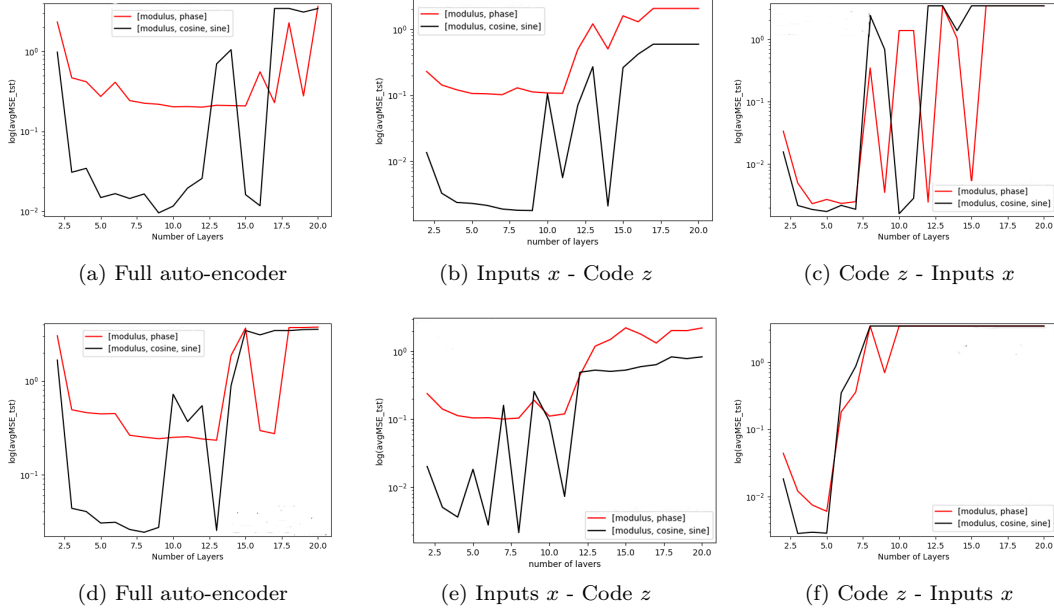


(a) Dataset of 10 features (b) Dataset of 20 features (c) Dataset of 10 features (d) Dataset of 20 features

Figure 3.4: Total MSE for activation functions ReLU, SELU, Sigmoid, and Tanh with $\nu = 8$. First two plots: the middle layer is trained to approximate Expression I of z . Expression II of z for last two plots.



(a) Dataset of 10 features (b) Dataset of 20 features (c) Dataset of 20 features (d) Dataset of 20 features
 Figure 3.5: MSE for each part of code z with activation function SELU ($\nu = 8$). First two plots: the middle layer is trained to approximate Expression I of z . Expression II of z for last two plots.



(a) Full auto-encoder (b) Inputs x - Code z (c) Code z - Inputs x
 (d) Full auto-encoder (e) Inputs x - Code z (f) Code z - Inputs x
 Figure 3.6: MSE for training the full auto-encoder (left), just the encoder (center), and just the decoder (right) with activation function SELU and $\nu = 8$. Plots on top show the results of the dataset of 10 features and 20 features on the bottom. Red lines show the results for evaluating Expression I of z and black lines for Expression II of Z .

3.2 Proposed Invariant Architecture and Its Experimental Validation

In this section, a new activation function f_{mixed} is defined in the sequence from Eq. 3.7 to 3.16, and it differs from regular activation functions such as ReLU or sigmoid.

O_1 in Eq. 3.7 takes the output from the previous layer of the layer where f_{mixed} is used. If f_{mixed} is used in the input layer, O_1 just takes the input data. Weight W and bias b in Eq. 3.8 are predefined and randomly generated from standard normal distribution. n_{col} is denoted as the number of columns/features. The dimension of W is $n_{col}(O_1) \times n_{col}(O_1)$ and $1 \times n_{col}(O_1)$ for b .

Eq. 3.12 and 3.13 are used to normalize X_{Even} (Eq. 3.9) and X_{Odd} (Eq. 3.10). The operations of O_2 , O_3 and O_4 are element-wise. The number of columns of O_2 , O_3 , O_4 , and O_5 are all half the number of columns of O_1 . For a regular activation function, the dimension of its output is the same as the dimension of inputs. For f_{mixed} , the number of outputs columns is increased because of the column-wise concatenation in Eq. 3.15. The total number of columns $n_{col}(outputs)$ of the output of f_{mixed} in Eq. 3.15 is $1 + 4 * 0.5 = 3$ times the number of columns of its input O_1 . Here, we denote output of each neuron from the same layer as columns.

$$f_{mixed} = \begin{cases} O_1 & = \text{Output of previous layer} & (3.7) \\ X & = O_1 W + b & (3.8) \\ X_{Even} & = \text{Choose the even columns of } X & (3.9) \\ X_{Odd} & = \text{Choose the odd columns of } X & (3.10) \\ O_2 & = \sqrt{X_{Even}^2 + X_{Odd}^2} & (3.11) \\ O_3 & = X_{Even} / O_2 \text{ (element-wise division)} & (3.12) \\ O_4 & = X_{Odd} / O_2 \text{ (element-wise division)} & (3.13) \\ O_5 & = O_3 * O_4 & (3.14) \\ Outputs & = [O_1, O_2, O_3, O_4, O_5] \text{ (column-wise concatenation)} & (3.15) \\ n_{col}(Outputs) & = 3 * n_{col}(O_1) & (3.16) \end{cases}$$

Suppose the previous layer in Eq. 3.7 consists of two neurons. O_1 takes the output of the previous layer which consists of two columns of input. We use $[input_1, input_2]$ two stand for input, where $input_1$ stand for the output of the first neuron and $input_2$ stand for the output of the second neuron from the previous layer. We use $O_{1,1}$ and $O_{1,2}$ to stand for the two columns of O_1 . Fig. 3.7 shows the plot $O_{1,1}$ vs $input_1$ and $O_{1,2}$ vs $input_2$ separately. As $O_{1,1} = input_1$ and $O_{1,2} = input_2$, the plots in Fig. 3.7a and Fig. 3.7b are just two straight lines. Fig. 3.8a, 3.8b, 3.8c and 3.8d show the plots of O_2 , O_3 , O_4 and O_5 vs two inputs $input_1$ and $input_2$.

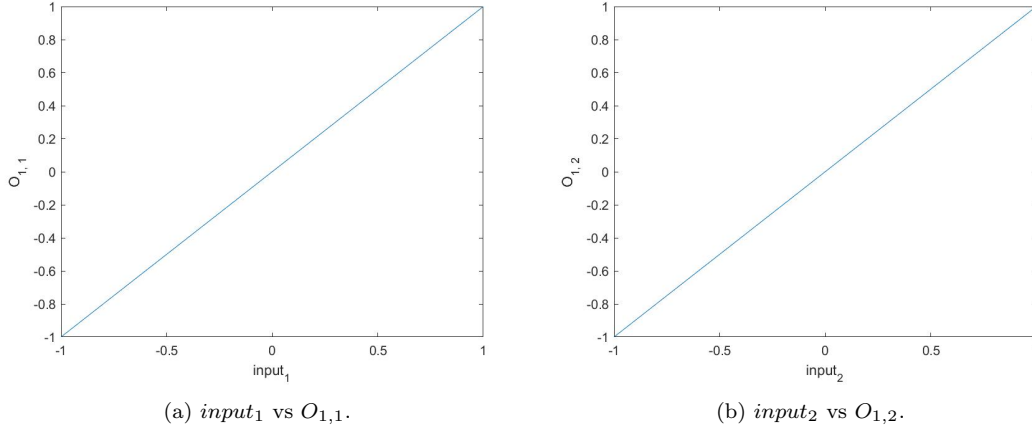
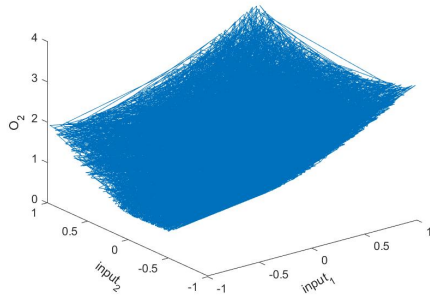
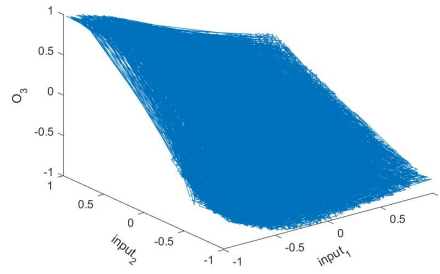


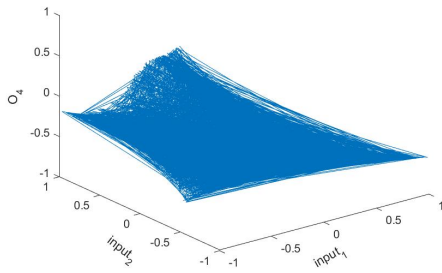
Figure 3.7: $input_1$ vs $O_{1,1}$ and $input_2$ vs $O_{1,2}$.



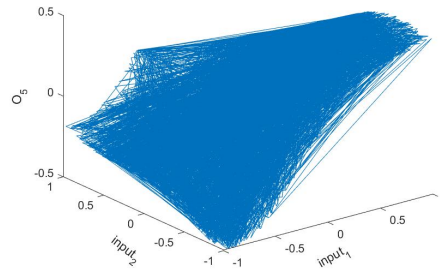
(a) $input_1$ and $input_2$ vs O_2 .



(b) $input_1$ and $input_2$ vs O_3 .



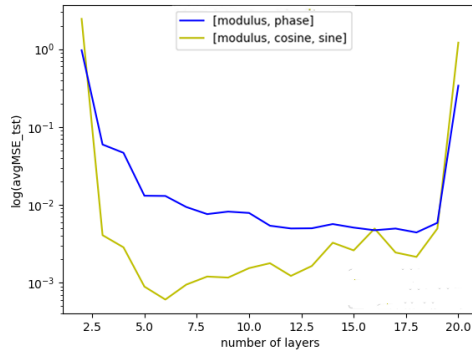
(c) $input_1$ and $input_2$ vs O_4 .



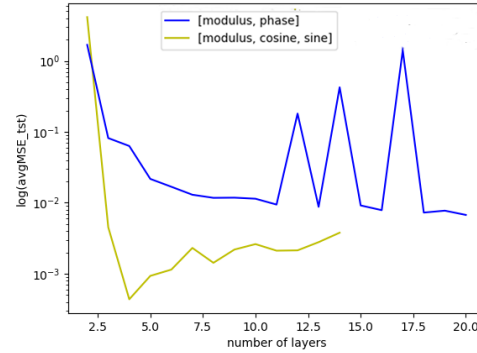
(d) $input_1$ and $input_2$ vs O_5 .

Figure 3.8: Plots of O_2 , O_3 , O_4 and O_5 .

The results in Figures 3.9a and 3.9b show that the new activation function is much more capable of approximating the invariance in the unrealistic case that the invariant representation, z^* , is known a priori. The MSE is lower by orders of magnitude.



(a) Dataset of 10 features



(b) Dataset of 20 features

Figure 3.9: Comparison of activation functions SELU and the new proposed activation function f_{mixed} for $\nu = 8$. Blue lines show the results for evaluating Expression I of z and yellow lines for Expression II of Z .

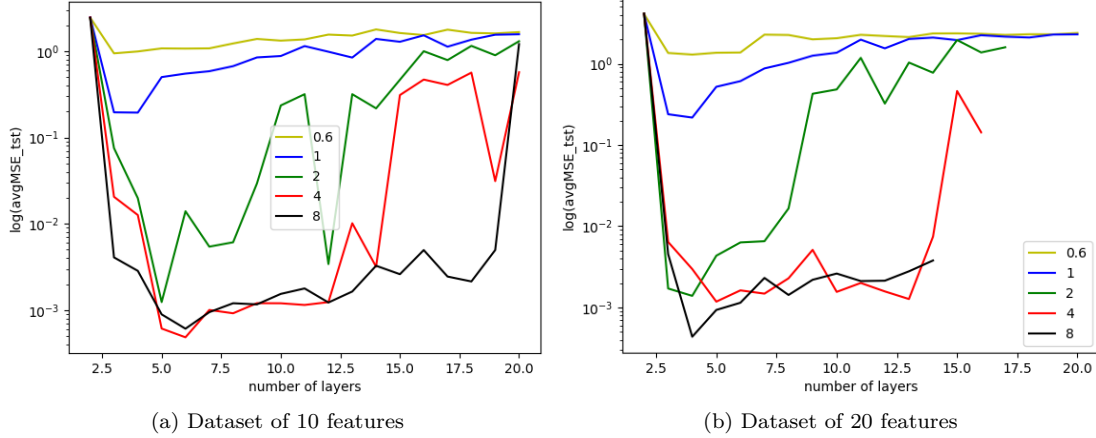


Figure 3.10: Total MSE in f_{mixed} by five different sizes of widths of the network hidden layers with $\nu \in [0.6, 1, 2, 4, 8]$.

3.3 Learning Invariant Representations from Relational Data

The experiments above assumed that we have a way of calculating the desired invariant representation z^* ; in the experiments above, we only considered translation invariance, for which z^* can be calculated by the discrete Fourier transform. That scenario is not practical: it virtually eliminates the need to learn invariance as a neural network, we can just use FFT.

The ultimate goal of our architecture is to receive information which samples are the same under some given model of invariance, and samples that are not the same, and train the network to discover on its own what the invariant transformation is. That is, we no longer have z^* defined for each x . In this way, the user is only required to know which inputs are the same, but is not required to analyze and mathematically specify what is the transformation behind the invariance.

To build a network that can learn invariance from data, we first define a cost function based on distances d^* between a pair of samples – the distance should take into account invariance, that is, it is null if one sample is an altered version of the other, and positive otherwise. We then present a network with pairs of samples, x , and x' , and the true value of d^* . We then train the auto-encoder $g(f(x))$, and we take part in the intermediate code, denote f_i , to learn invariance. We train the network so that the intermediate code resulting from the $z = f_i(x)$ and $z' = f_i(x')$ will have $d = \|z - z'\|$ similar to d^* . We are most interested in preserving small distances; thus we use the inverse of squared Euclidean distance as below,

$$inv(d) = \frac{1}{\delta + \alpha d} \quad (3.17)$$

where δ and α are distance parameters. We then compared the true and the actual inverse distance using squared error. Thus, the final invariance term in the objective function is

$$\Omega(z, z^*) = (inv(d) - inv(d^*))^2. \quad (3.18)$$

3.4 Experimental Validation of Learning Invariant Representations from Data

We conducted a series of experiments to validate the ability of the new activation function to learn invariance from data and used Expression II of z in the experiments. We changed the number of network layers in range of 3 and 10.

Our first experiment involves learning translation invariance from data. We create a dataset in which samples come in pairs, the first sample is random as described before, and the second sample is a shifted version of the first sample, with the amount of shift selected randomly. For example, if we have a sample 1, 2, 3, 4, 5, and we want to shift this sample by 2 positions, this sample is then changed to 4, 5, 1, 2, 3. As the true distance, d^* we use the Euclidean distance between modulus of Fourier transforms of the samples; thus, we have null distance if two samples are shifted versions of each other, and positive distance otherwise.

The results presented in Figure 3.11 show that in the absence of the true desired values of the intermediate code, and with access to pairwise distance data instead, the auto-encoder is still able to learn invariant representation equivalent to the Fourier transform of the input.

To move beyond simple shift-invariance, we created a dataset in which each sample compose of two parts, left and right, and a circular shift occurs independently within each part. The parts are of equal size, that is, if the sample has 10 features, each part consists of 5 features. If one or both parts of the sample are shifted version of another sample, the true distance d^* is null. We also create samples which are the same, concerning invariance, only in one part – those samples have $d^* > 0$ and allow us to detect if invariance for both parts is appropriately learned.

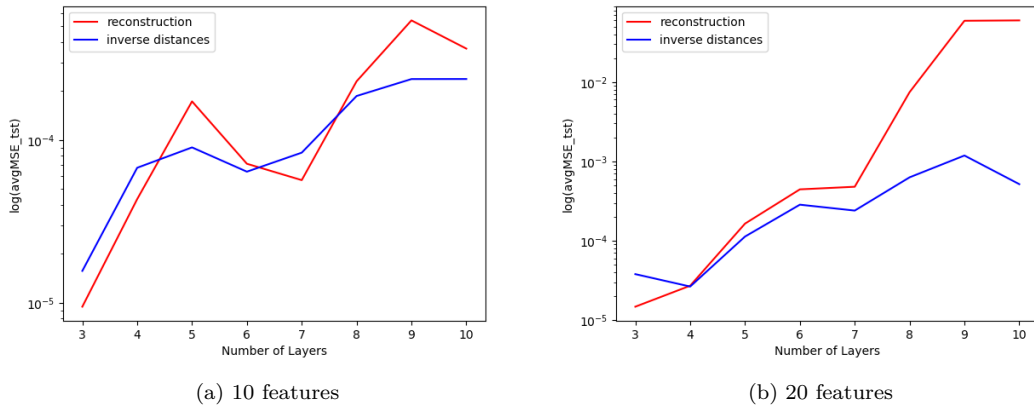


Figure 3.11: MSE for reconstruction and for approximating the inverse distances for $\nu = 8$.

The results in Figure 3.12 show that our architecture can successfully learn this type of invariance – the MSE is below 10^{-4} . We also conducted an experiment in which the left and right parts are of different sizes: 3 and 7, or 5 and 15 (Fig. 3.13).

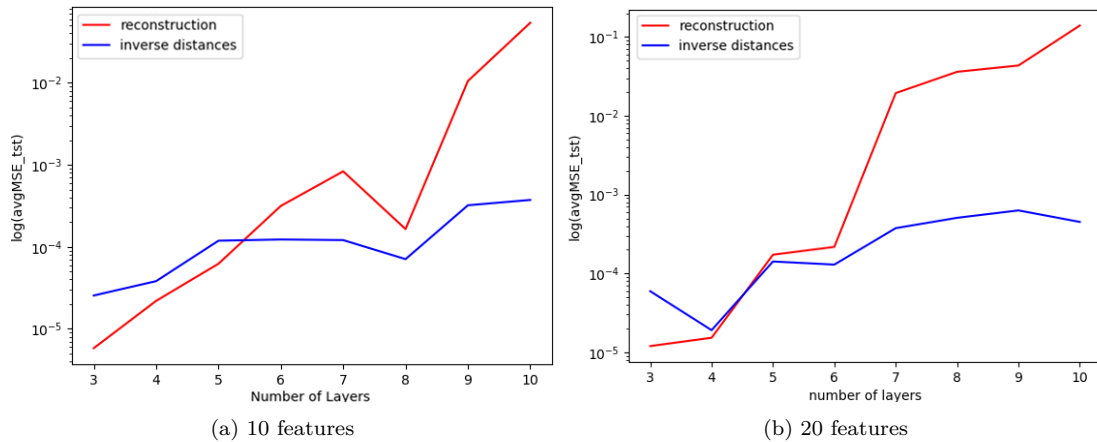


Figure 3.12: MSE for reconstruction and for approximating the inverse distances for two-part samples.

In the case of the left part and right part have the same number of features, we changed the order of columns, so that the two parts are not contiguous blocks of features, but instead follow an interlaced pattern of $L_1R_1L_2R_2\dots$ (Fig. 3.14; L_i : the i -th feature from the left part; R_i : the i -th feature from the right part), or a random pattern (Fig. 3.15; the position of all the features are randomly permuted). In this way, the transformation to which the neural network should be invariant is no longer a variant of translation, but an arbitrary permutation instead. The results again indicate the the proposed architecture is effective in learning the invariance.

We also analyzed if the new architecture can learn invariance to scale, not just to shifts. In Fig. 3.16, we show that it can learn invariance to multiplying the input by a scalar, and in Fig. 3.17, invariance to both shifting and scalar multiplication.

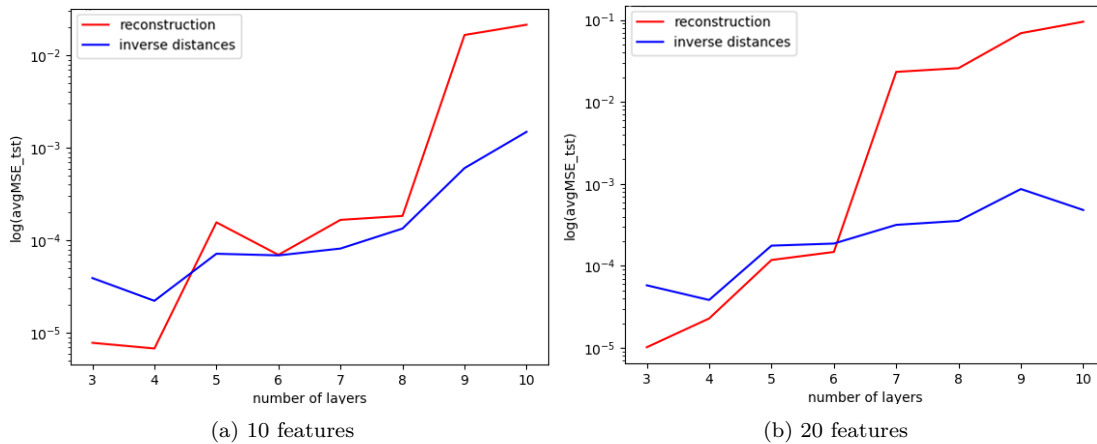


Figure 3.13: MSE for reconstruction and for approximating the inverse distances for two-uneven-parts samples.

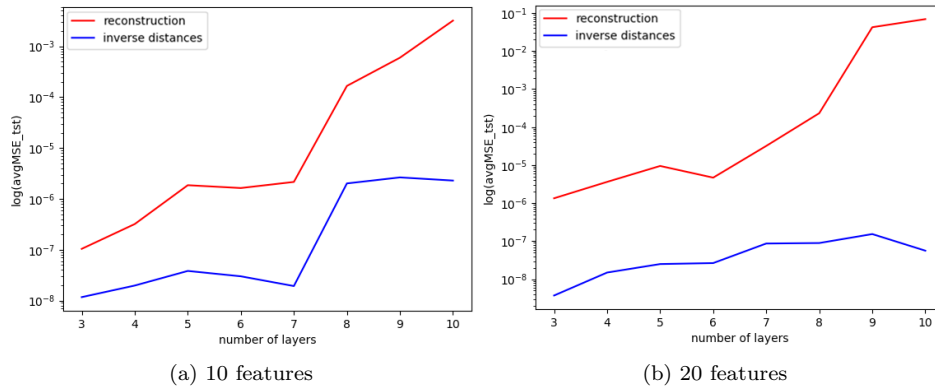


Figure 3.14: MSE for reconstruction and for approximating the inverse distances for interlaced two-part samples.

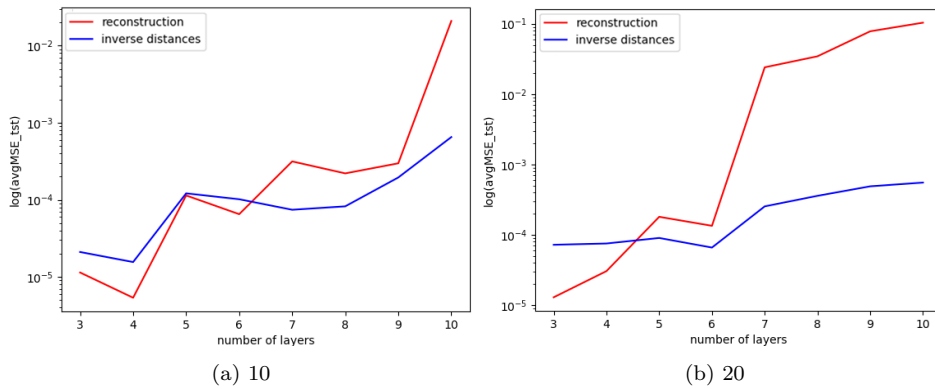


Figure 3.15: MSE for reconstruction and for approximating the inverse distances for randomly-shuffled columns, two-part samples.

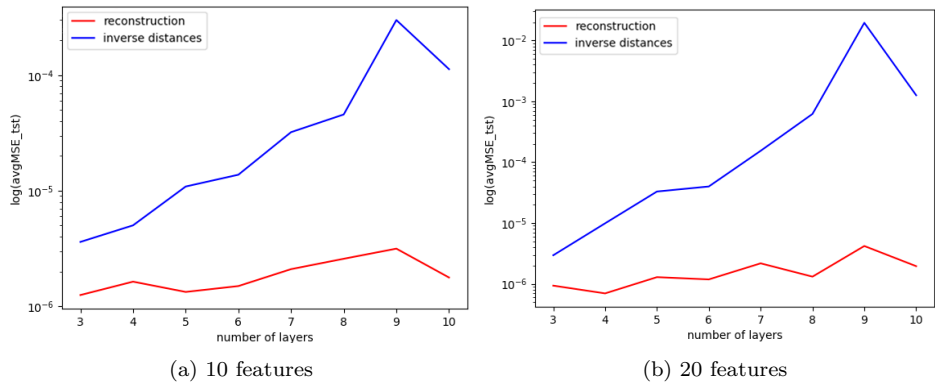


Figure 3.16: MSE of reconstruction and inverse distances for invariance to scalar multiplication.

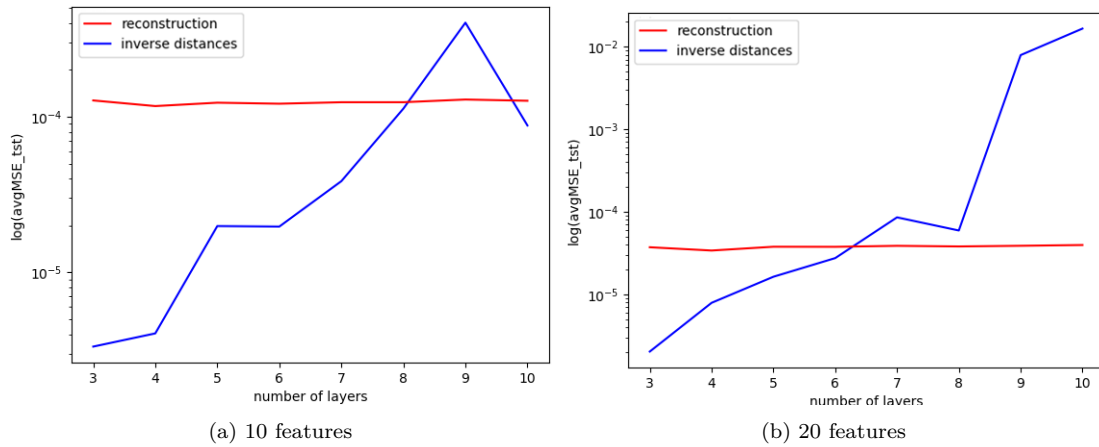


Figure 3.17: MSE of reconstruction and inverse distances for invariance to both shifting and scalar multiplication.

In summary, we have conducted two parts of the experiments. The first part was designed to show the proposed activation function is more accurate than traditional architectures in approximating one specific transformation: the transform from spatial to frequency domain resulting from the discrete Fourier transform. This transform was chosen because it is invariant to translation of the input. The experiments in the second part were designed to show that our new network architecture can be used to learn invariance to different transformations: shifting, scaling, and randomly shuffling, and it can still achieve excellent performance even if the desired output of the invariant transformation is not known a priori. Overall, the experiments show that our proposed new approach is capable of learning complex invariances from data, even if we do not know what specific form of invariance is present in the data.

Chapter 4

Learning Invertible Neural Blocks

Neural networks learn to approximate a mapping $x \rightarrow y$ from data. Typically, such mapping is not invertible – some information is lost as the input is processed through the neural network, and the output does not have full information that would allow for reconstructing x from y . Recently, techniques for ensuring that the neural network preserves all information have attracted attention, resulting in invertible neural networks: the network can guarantee that if it learns a mapping $x \rightarrow y$, the inverse mapping $y \rightarrow x$ exists. Ability to train a mapping that is guaranteed to be invertible has practical applications; for example, they give rise to normalizing flows [34, 35], which allow for sampling from a complicated, multi-modal probability distribution by generating samples from a simple one, and transforming them through an invertible mapping. For any given architecture for invertible neural networks, it is important to know whether it can be trained to approximate arbitrary invertible mappings, or its approximation capabilities are limited.

4.1 Architectures for Invertible Neural Networks

Here, we analyze approximation capabilities of two recently proposed invertible architectures, Neural ODEs (ODE-Nets) [23], and i-ResNets [22]. Here, we show that both of these models are limited in their approximation capabilities, and then prove that with a single simple technique, this limitation can be removed.

4.1.1 Neural ODEs

Neural ODEs [23] are a recently proposed class of differentiable neural network building blocks that lead to invertible models. ODE-Nets were formulated by observing that processing an initial input vector x_0 through a sequence of residual blocks can be seen as an evolution of x_t in time $t \in [T]$, where $[T] = \{1, \dots, T\}$. Then, a residual block (eq. (4.1)) is a discretization of a continuous-time system of ordinary differential equations (eq. (4.2))

$$x_{t+1} - x_t = f_{\Theta}(x_t, t), \quad (4.1)$$

$$\frac{dx_t}{dt} = \lim_{\delta_t \rightarrow 0} \frac{x_{t+\delta_t} - x_t}{\delta_t} = f_{\Theta}(x_t, t). \quad (4.2)$$

The transformation $\phi_T : \mathcal{X} \rightarrow \mathcal{X}$ taking x_0 into x_T realized by an ODE-Net for some chosen, fixed time $T \in \mathbb{R}$ is not specified directly through a functional relationship $x \rightarrow f(x)$ for some neural network f , but indirectly, through the solutions to the initial value problem (IVP) of the ODE

$$x_T = \phi_T(x_0) = x_0 + \int_0^T f_{\Theta}(x_t, t) dt \quad (4.3)$$

involving some underlying neural network $f_{\Theta}(x_t, t)$ with trainable parameters Θ . By a *p*-ODE-Net we denote an ODE-Net that takes a *p*-dimensional sample vector on input and produces a *p*-dimensional vector on output. The underlying network f_{Θ} must match those dimensions on its input and output, but in principle can have arbitrary internal architecture. The adjoint sensitivity method [36] based on reverse-time integration of an expanded ODE allows for finding gradients of the IVP solutions $\phi_T(x_0)$ with respect to parameters Θ and the initial values x_0 . This enables training ODE-Nets to use gradient descent, as well as combining them with other neural network blocks. Benefits of ODE-Nets compared to residual blocks include improved memory and parameter efficiency, ease of modeling phenomena with continuous-time dynamics, out-of-the-box invertibility ($x_0 = \phi_{-T}(x_T)$), and simplified computations of normalizing flows [23]. Since their introduction, ODE-Nets have seen improved implementations [37] and enhancements in training and stability [38, 39]. The question of their approximation capabilities remains, however, unresolved.

4.1.2 Invertible Residual Networks

While ResNets refer to arbitrary networks with any residual blocks $x_{t+1} = x_t + f_{\Theta}(x_t, t)$, that is, can have any residual mapping $f_{\Theta}(x_t, t)$, i-ResNets [22], and their improved variant, Residual Flows [40], are built from blocks in which f_{Θ} is Lipschitz-continuous with constant lower than 1 as a function of x_t for fixed t , which we denote by $\text{Lip}(f_{\Theta}) < 1$. This simple constraint is sufficient [22] to guarantee invertibility of the residual network, that is, to make $x_t \rightarrow x_{t+1}$ a one-to-one mapping.

Given the constraint on the Lipschitz constant, an invertible mapping $x \rightarrow 2x$ cannot be performed by a single i-ResNet layer. But a stack of two layers, each of the form $x \rightarrow x + (\sqrt{2} - 1)x$ and thus Lipschitz-continuous with constant lower than 1, yields the desired mapping. A single i-ResNet layer $x_{t+1} = (I + f_{\Theta})(x_t, t)$, where I is the identity mapping, is $\text{Lip}(I + f_{\Theta}) = k < 2$, and a composition of T such layers has Lipschitz constant of at most $K = k^T$. Thus, for any finite K , it might be possible to approximate any invertible mapping h with $\text{Lip}(h) \leq K$ by a series of i-ResNet layers, with the number of layers depending on K . However, the question whether the possibility signaled above is true, and i-ResNet have universal approximation capability within the class of invertible continuous mappings, has not been considered thus far.

4.1.3 Limitations of Approximation Capabilities of Neural ODEs and i-ResNets

A Neural ODE on its own does not have universal approximation capability. Consider a continuous, differentiable, invertible function $f(x) = -x$ on $\mathcal{X} = \mathbb{R}$. There is no ODE defined on \mathbb{R} that would result in $x_T = \phi_T(x_0) = -x_0$. Informally, in ODEs, paths (x_t, t) between the initial value $(x_0, 0)$ and final value (x_T, T) have to be continuous and cannot intersect in $\mathcal{X} \times \mathbb{R}$ for two different initial values. Paths corresponding to $x \rightarrow -x$ and $0 \rightarrow 0$ would need to intersect. These observations have recently been illustrated by the authors of ANODE [41], who also show empirical evidence

indicating that expanding the dimensionality and using q -ODE-Net for $q > p$ instead of a p -ODE-Net has a positive impact on the training of the model and on its generalization capabilities.

Here, we prove that setting $q = p+1$ is enough to turn Neural ODE followed by a linear layer into a universal approximator. Next, we focus our attention on invertible functions – homeomorphisms – by exploring pure p -ODE-Nets, not capped by a linear layer. We go beyond the $x \rightarrow -x$ example and show a class of $\mathcal{X} \rightarrow \mathcal{X}$ invertible mappings that cannot be expressed by Neural ODEs defined on \mathcal{X} . Our main result is a proof that any homeomorphism $\mathcal{X} \rightarrow \mathcal{X}$, for $\mathcal{X} \subset \mathbb{R}^p$, can be modeled by a Neural ODE operating on a Euclidean space of dimensionality $2p + 1$ that embeds \mathcal{X} as a linear subspace. We show a similar result for i-ResNets.

4.2 Augmented Neural ODEs are Universal Approximators

We show, through a simple construction, that a Neural ODE followed by a linear layer can approximate functions equally well as any traditional feed-forward neural network. Since networks with shallow-but-wide fully-connected architecture [42, 43], or narrow-but-deep ResNet-based architecture [44] are universal approximators, so are ODE-Nets.

Theorem 4.2.1. *Consider a neural network $F : \mathbb{R}^p \rightarrow \mathbb{R}$ that approximates a Lebesgue integrable function $f : \mathcal{X} \rightarrow \mathbb{R}$, with $\mathcal{X} \subset \mathbb{R}^p$ being a compact subset. For any $q = p + r$, $r \geq 1$, there exists a linear layer-capped q -ODE-Net that can perform the mapping F .*

Proof. Set $r = 1$. Let G be a neural network that takes input vectors $x^{(q)} = [x^{(p)}, x^{(r)}]$ and produces q -dimensional output vectors $y^{(q)} = [y^{(p)}, y^{(r)}]$, where $y^{(r)} = F(x^{(p)})$ is the desired transformation. G is constructed as follows: use F to produce $y^{(r)} = F(x^{(p)})$, ignore $x^{(r)}$, and always output $y^{(p)} = 0$. Consider a q -ODE-Net defined through $dx/dt = G(x_t) = [0^{(p)}, F(x_t^{(p)})]$. Let the initial value be $x_0 = [x^{(p)}, 0^{(r)}]$. The ODE will not alter the first p dimensions throughout time, hence for any t , $F(x_t^{(p)}) = y^{(r)}$. After time $T = 1$, we will have

$$x_T = x_0 + \int_0^1 G(x_t) dt = [x^{(p)}, 0^{(r)}] + \int_0^1 [0^{(p)}, y^{(r)}] dt = [x^{(p)}, F(x^{(p)})].$$

Thus, for any $x \in \mathbb{R}^p$, the output $F(x)$ can be recovered from the output of the ODE-Net by a simple, sparse linear layer that ignores all dimensions except the last one, which it returns. \square

ODE-Nets have two main advantages compared to traditional architectures: improved computational and space efficiency, and out-of-the-box invertibility. The construction above nullifies both and thus is of theoretical interest only. This introduces two new open problems: can Neural ODEs be universal approximators while showing improved efficiency compared to traditional architectures, and can Neural ODEs model any invertible function h , assuming h and h^{-1} are continuous. The main focus of this work is to address the second problem.

4.3 Background on ODEs, Flows, and Embeddings

4.3.1 Correspondence between Flows and ODEs

Given a continuous flow $(\mathcal{X}, \mathbb{R}, \Phi)$ one can define a corresponding ODE operating on \mathcal{X} by defining a vector $V(x) \in \mathbb{R}^p$ for every $x \in \mathcal{X} \subset \mathbb{R}^p$ such that $V(x) = d_x \Phi(x, t)/dt|_{t=0}$. Then, the ODE

$$\frac{dx_t}{dt} = V(x_t), \quad (4.4)$$

$$\phi_{(T-S)}(x_S) = x_S + \int_S^T V(x_t) dt, \quad (4.5)$$

corresponds to continuous flow Φ . $V(x_t)$ starts at time S and ends at time T . Indeed, ϕ_0 is identity, and $\phi_{(S+T)} = \phi_S + \phi_T$. Thus, for any homeomorphism family Φ defining a continuous flow, there is a corresponding ODE that, integrated for time T , models the flow at time T , $\phi_T(x)$. The vectors of derivatives $V(x) \in \mathbb{R}^p$ for all $x \in \mathcal{X}$ are continuous over \mathcal{X} and are constant in time, and define a *continuous vector field* over \mathbb{R}^p . The ODEs are evolving according to such a time-invariant vector field, where the right-hand side of Eq. 5.2 depends on x_t but not directly on time t , are called *autonomous ODEs*, and take the form of $dx/dt = f_\Theta(x_t)$. Any *time-dependent ODE* (eq. (5.2)) can be transformed into an autonomous ODE by removing time t from being a separate argument of $f_\Theta(x_t, t)$, and adding it as part of the vector x_t . Specifically, we add an additional dimension $x[\tau]$ to vector x , with $\tau = p + 1$. We equate it with time, $x[\tau] = t$, by including $dx[\tau]/dt = 1$ in the definition of how f_Θ acts on x_t , and including $x_0[\tau] = 0$ in the initial value x_0 . In defining f_Θ , explicit use of t as a variable is being replaced by using the component $x[\tau]$ of vector x_t . The result is an autonomous ODE. Given time T and an ODE defined by f_Θ , ϕ_T , the flow at time T , may not be well defined, for example, if f_Θ diverges to infinity along the way. However, if f_Θ is well behaved, the flow will exist at least locally around the initial value. Specifically, Picard–Lindelöf theorem states that if an ODE is defined by a Lipschitz-continuous function $f_\Theta(x_t)$, then there exists $\varepsilon > 0$ such that the flow at time T , ϕ_T , is well-defined and unique for $-\varepsilon < T < \varepsilon$. If exists, ϕ_T is a homeomorphism, since the inverse exists and is continuous; simply, ϕ_{-T} is the inverse of ϕ_T .

4.3.2 Flow Embedding Problem for Homeomorphisms

Given a p -flow, we can always find a corresponding ODE. Given an ODE, under mild conditions, we can find a corresponding flow at time T , ϕ_T , and it necessarily is a homeomorphism. Is the class of p -flows equivalent to the class of p -homeomorphisms, or only to its subset? That is, given a homeomorphism h , does a p -flow such that $\phi_T = h$ exist? This question is referred to as the problem of embedding the homeomorphism into a flow.

For a homeomorphism $h : \mathcal{X} \rightarrow \mathcal{X}$, its *restricted embedding into a flow* is a flow $(\mathcal{X}, \mathbb{R}, \Phi)$ such that $h(x) = \Phi(x, T)$ for some T ; the flow is restricted to be on the same domain as the homeomorphism. Studies of homeomorphisms on simple domains such as a 1D segment [45] or a 2D plane [46] already showed that a restricted embedding not always exists.

An *unrestricted embedding into a flow* [47] is a flow $(\mathcal{Y}, \mathbb{R}, \Phi)$ on some space \mathcal{Y} of dimensionality higher than p . It involves a homeomorphism $g : \mathcal{X} \rightarrow \mathcal{Z}$ that maps \mathcal{X} into some subset $\mathcal{Z} \subset \mathcal{Y}$, such that the flow on \mathcal{Y} results in mappings on \mathcal{Z} that are equivalent to h on \mathcal{X} for some T , that is, $g(h(x)) = \Phi(g(x), T)$. While a solution to the unrestricted embedding problem always exists, it

involves a smooth, non-Euclidean manifold \mathcal{Y} . For a homeomorphism $h : \mathcal{X} \rightarrow \mathcal{X}$, the manifold \mathcal{Y} , variously referred to as the twisted cylinder [47], or a suspension under a ceiling function [48], or a mapping torus [49], is a quotient space $\mathcal{Y} = \mathcal{X} \times [0, 1] / \sim$ defined through the equivalence relation $(x, 1) \sim (h(x), 0)$. The flow that maps x at $t = 0$ to $h(x)$ at $t = 1$ and $h(h(x))$ at $t = 2$ involves trajectories in $\mathcal{X} \times [0, 1] / \sim$ in the following way: for t going from 0 to 1, the trajectory tracks in a straight line from $(x, 0)$ to $(x, 1)$, which in the quotient space is equivalent to $(h(x), 0)$. Then, for t going from 1 to 2, the trajectory proceeds from $(h(x), 0)$ to $(h(x), 1) \sim (h(h(x)), 0)$. The fact that the solution to the unrestricted embedding problem involves a flow on a non-Euclidean manifold makes applying it in the context of gradient-trained ODE-Nets difficult.

4.4 Approximation of Homeomorphisms by Neural ODEs

In exploring the approximation capabilities of Neural ODEs for p -homeomorphisms, we will assume that the neural network $f_{\Theta}(x_t)$ on the right-hand side of the ODE is a universal approximator and thus can be made large enough to approximate arbitrary function arbitrarily well. Thus, our concern is with what flows can be modeled, assuming ODE-Net can have arbitrary internal dimensionality, depth, and architecture. We only care about the input-output dimensionality q of the q -ODE-Net. We consider two scenarios, $q = p$, and $q > p$.

4.4.1 Restricting the Dimensionality Limits Capabilities of Neural ODEs

We show a class of functions that a Neural ODE cannot model, a class that generalizes the $x \rightarrow -x$ one-dimensional example.

Theorem 4.4.1. *Let $\mathcal{X} = \mathbb{R}^p$, and let $\mathcal{Z} \subset \mathcal{X}$ be a set that partitions \mathcal{X} into two or more disjoint, connected subsets C_i , for $i = [m]$. Consider a mapping $h : \mathcal{X} \rightarrow \mathcal{X}$ that*

- *is an identity transformation on \mathcal{Z} , that is, $\forall z \in \mathcal{Z}, h(z) = z$,*
- *maps some $x \in C_i$ into $h(x) \in C_j$, for $i \neq j$.*

Then, no p -ODE-Net can model h .

Proof. A p -ODE-Net can model h if a restricted flow embedding of h exists. Suppose that it does, a continuous flow $(\mathcal{X}, \mathbb{R}, \Phi)$ can be found for h such that the trajectory of $\Phi(x, t)$ is continuous on $t \in [0, T]$ with $\Phi(x, 0) = x$ and $\Phi(x, T) = h(x)$ for some $T \in \mathbb{R}$, for all $x \in \mathcal{X}$.

If h maps some $x \in C_i$ into $h(x) \in C_j$, for $i \neq j$, the trajectory from $\Phi(x, 0) = x \in C_i$ to $\Phi(x, T) = h(x) \in C_j$ crosses \mathcal{Z} – there is $z \in \mathcal{Z}$ such that $\Phi(x, \tau) = z$ for some $\tau \in (0, T)$. From uniqueness and reversibility of ODE trajectories, we then have $\Phi(z, -\tau) = x$. From additive property of flows, we have $\Phi(z, T - \tau) = h(x)$.

Since h is identity over \mathcal{Z} and $\mathcal{Z} \subset \mathcal{X}$, thus $h(z) = \Phi(z, T) = \Phi(z, 0) = z$. That is, the trajectory over time T is a closed curve starting and ending at z , and $\Phi(z, t) = \Phi(z, T + t)$ for any $t \in \mathbb{R}$. Specifically, $\Phi(z, T - \tau) = \Phi(z, -\tau) = x$. Thus, $h(x) = x$. We arrive at a contradiction with the assumption that x and $h(x)$ are in two disjoint subsets of \mathbb{R}^p separated by \mathcal{Z} . Thus, no p -ODE-Net can model h . \square

The result above shows that Neural ODEs applied in the most natural way, with $q = p$, are severely restricted in the way distinct regions of the input space can be rearranged in order to learn and generalize from the training set, and the restrictions go well beyond requiring invertibility and continuity.

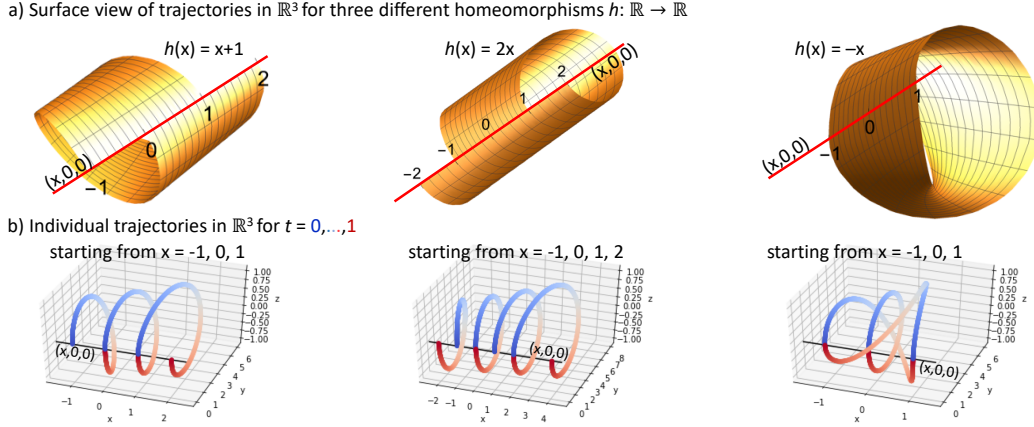


Figure 4.1: Proposed flow in \mathbb{R}^{2p+1} that embeds an $\mathbb{R}^p \rightarrow \mathbb{R}^p$ homeomorphism. Three examples for $p = 1$ are shown from left to right, including the mapping $h(x) = -x$ that cannot be modeled by a flow on \mathbb{R}^p , but can in \mathbb{R}^{2p+1} .

4.4.2 Neural ODEs with Extra Dimensions are Universal Approximators for Homeomorphisms

If we allow the Neural ODE to operate on Euclidean space of dimensionality $q > p$, we can approximate arbitrary p -homeomorphism $\mathcal{X} \rightarrow \mathcal{X}$, as long as q is high enough. Here, we show that suffices to take $q = 2p + 1$. We construct a mapping from the original problem space, $\mathcal{X} \in \mathbb{R}^p$ into \mathbb{R}^{2p+1} that

- preserves \mathcal{X} as a p -dimensional linear subspace consisting of vectors $[x, 0^{(p+1)}]$,
- leads to an ODE on \mathbb{R}^{2p+1} that maps $[x, 0^{(p+1)}] \rightarrow [h(x), 0^{(p+1)}]$.

Thus, we provide a solution with a structure that is convenient for out-of-the-box training and inference using Neural ODEs – it is sufficient to add $p + 1$ dimensions, all zeros, to the input vectors. Our main result is the following.

Theorem 4.4.2. *For any homeomorphism $h: \mathcal{X} \rightarrow \mathcal{X}$, $\mathcal{X} \subset \mathbb{R}^p$, there exists a $(2p + 1)$ -ODE-Net $\phi_T: \mathbb{R}^{2p+1} \rightarrow \mathbb{R}^{2p+1}$ for $T = 1$ such that $\phi_T([x, 0^{(p+1)}]) = [h(x), 0^{(p+1)}]$ for any $x \in \mathcal{X}$.*

Proof. We prove the existence in a constructive way, by showing a vector field in \mathbb{R}^{2p+1} , and thus an ODE, with the desired properties. Let $\delta_x, z_x \in \mathbb{R}^p$ be defined as

$$\delta_x = h(x) - x, \quad z_x = r(x),$$

where $r: \mathbb{R} \rightarrow \mathbb{R}_+$ is bounded away from zero, and is a smooth, strictly monotonic function. It is applied to a vector entry-wise; in Fig. 4.1 we used $z_x[i] = \log(1 + \exp(x[i] + 2))$.

We start with the extended space (x, τ) with a variable τ corresponding to time added as the last dimension, as in the construction of an autonomous ODE from time-dependent ODE. We then define a mapping $y(x, \tau): \mathbb{R}^p \times \mathbb{R} \rightarrow \mathbb{R}^{2p+1}$. For $\tau \in [0, 1]$, the mapping (see Fig. 4.1) is defined through

$$y(x, \tau) = \left[x + \frac{1 - \cos \pi \tau}{2} \delta_x, z_x(1 - \cos 2\pi \tau), \sin 2\pi \tau \right]. \quad (4.6)$$

The mapping indeed just adds $p + 1$ dimensions of 0 to x at time $\tau = 0$, and at time $\tau = 1$ it gives the result of the homeomorphism applied to x , again with $p + 1$ dimensions of 0

$$\begin{aligned} y(x, 0) &= [x, 0^{(p)}, 0], \\ y(x, 1) &= [x + \delta_x, 0^{(p)}, 0] = [h(x), 0^{(p)}, 0] = y(h(x), 0). \end{aligned}$$

We can use these properties to define the mapping for $\tau \notin [0, 1]$, by setting $y(x, \tau) = y(h^{(\lfloor \tau \rfloor)}, \tau - \lfloor \tau \rfloor)$; for example, $y(x, -1.75) = y(h^{-1}(h^{-1}(x)), 0.25)$. Intuitively, the mapping $y(x, \tau)$ will provide the position in \mathbb{R}^{2p+1} of the time evolution for duration τ of an ODE on \mathbb{R}^{2p+1} starting from a position corresponding to x . For $x \neq x'$, for any given τ , we have $y(x, \tau) \neq y(x', \tau)$, since $x \rightarrow z_x$ is a one-to-one mapping – it was defined by a strictly monotonic function r . Thus, in \mathbb{R}^{2p+1} , paths starting from two distinct points do not intersect at the same point in time. Intuitively, we have added enough dimensions to the original space so that we can reroute all trajectories without intersections.

We have τ correspond directly to time, that is, $d\tau/dt = 1$ and $\tau = 0$ for $t = 0$. The mapping y has continuous derivative with respect to t , defining a vector field over the image of y , a subset of \mathbb{R}^{2p+1}

$$\frac{dy}{dt} = \left[\frac{\pi \delta_x}{2} \sin \pi t, 2\pi z_x \sin 2\pi t, 2\pi \cos 2\pi t \right].$$

We can verify that the vector field defined through derivatives of $y(x, t)$ with respect to time has the same values for $t = 0$ and $t = 1$ for any x

$$\frac{dy}{dt}(x, 0) = [0^{(p)}, 0^{(p+1)}, 2\pi], \quad \frac{dy}{dt}(x, 1) = [0^{(p)}, 0^{(p+1)}, 2\pi],$$

Thus,

$$\frac{dy}{dt}(x, 1) = \frac{dy}{dt}(h(x), 0),$$

the vector field is well-behaved at $y(x, 1) = y(h(x), 0)$ – it is continuous over the whole image of y . The vector field above is defined over a closed subset $y(x, \tau)$ of \mathbb{R}^{2p+1} , and can be (see [50], Lemma 8.6) extended to the whole \mathbb{R}^{2p+1} . A $(2p + 1)$ -ODE-Net with a universal approximator network f_Θ on the right hand side can be designed to approximate the vector field arbitrarily well. The resulting ODE-Net approximates $[x, 0^{p+1}]$ to $[h(x), 0^{p+1}]$. \square

Based on the above result, we now have a simple method for training a Neural ODE to approximate a given continuous, invertible mapping h and, for free, also obtain its continuous inverse h^{-1} . On input, each sample x is augmented with $p + 1$ zeros. For a given x , the output of the ODE-Net is split into two parts. The first p dimensions are connected to a loss function that penalizes the deviation from $h(x)$. The remaining $p + 1$ dimensions are connected to a loss function that penalizes for any deviation from 0. Once the network is trained, we can get h^{-1} by using an ODE-Net with $-f_\Theta$ instead of f_Θ used in the trained ODE-Net.

4.5 Approximation of Homeomorphisms by i-ResNets

4.5.1 Restricting the Dimensionality Limits Capabilities of i-ResNets

We show that similarly to Neural ODEs, i-ResNets cannot model a simple $f(x) \rightarrow -x$ homeomorphism $\mathbb{R} \rightarrow \mathbb{R}$, indicating that their approximation capabilities are limited.

Theorem 4.5.1. *Let $F_n(x) = (I + f_n) \circ (I + f_{n-1}) \circ \dots \circ (I + f_1)(x)$ be an n -layer i-ResNet, and let $x_0 = x$ and $x_n = F_n(x_0)$. If $\text{Lip}(f_i) < 1$ for all $i = 1, \dots, n$, then there are no number $n \geq 1$ and no functions f_i for all $i = 1, \dots, n$ such that $x_n = -x_0$.*

Proof. Consider $a_0 \in \mathbb{R}$ and $b_0 = a_0 + \delta_0$. Then, $a_1 = a_0 + f_1(a_0)$ and $b_1 = a_0 + \delta_0 + f_1(a_0 + \delta_0)$. From $\text{Lip}(f_1) < 1$ we have that $|f_1(a_0 + \delta_0) - f_1(a_0)| < |\delta_0|$. Let $\delta_1 = b_1 - a_1$. Then, we have

$$\begin{aligned} \delta_1 &= a_0 + \delta_0 + f_1(a_0 + \delta_0) - a_0 - f_1(a_0) \\ &= \delta_0 + f_1(a_0 + \delta_0) - f_1(a_0), \\ \delta_1 &> \delta_0 - |\delta_0|, \\ \delta_1 &< \delta_0 + |\delta_0|. \end{aligned}$$

That is, δ_1 has the same sign as δ_0 . Thus, applying the reasoning to arbitrary i , $i + 1$ instead of $0, 1$, if $a_i < b_i$, then $a_{i+1} < b_{i+1}$, and if $a_i > b_i$, then $a_{i+1} > b_{i+1}$, for all $i = 0, \dots, n - 1$. Assume we can construct an i-ResNet F_n such that $F_n(0) = 0$; then $F_n(x) > 0$ for any $x > 0$, and $F_n(x)$ cannot map x into $-x$. \square

The result above leads to a more general observation about paths in spaces of dimensionality higher than one. As with ODE-Nets, we will use p -i-ResNet to denote an i-ResNet operating on \mathbb{R}^p .

Corollary 4.5.1. *Let the straight line connecting $x_t \in \mathbb{R}^p$ to $x_{t+1} = x_t + f(x_t) \in \mathbb{R}^p$ be called an extended path $x_t \rightarrow x_{t+1}$ of a time-discrete topological transformation group on $\mathcal{X} \in \mathbb{R}^p$. In p -i-ResNet, for $x_t \neq x'_t$, extended paths $x_t \rightarrow x_{t+1}$ and $x'_t \rightarrow x'_{t+1} = x'_t$ do not intersect.*

Proof. For two extended paths to intersect, vectors $x_t, x'_t, x_{t+1}, x'_{t+1}$ have to be co-planar. If we restrict attention to dynamics starting from x_t, x'_t , we can view it as a one-dimensional system, with the space axis parallel to $x_t - x'_t$, and time axis orthogonal to it. If $x_{t+1} - x'_{t+1}$ is parallel to $x_t - x'_t$, Theorem 4.5.1 shows that if x'_t is above x_t , then x'_{t+1} is above x_{t+1} ; extended paths do not intersect.

If $x_{t+1} - x'_{t+1}$ is not parallel to $x_t - x'_t$, construct a new i-ResNet $x \rightarrow x + g(x)$ with $g(x_t) = cf(x_t)$ and $g(x'_t) = c'f(x'_t)$, with $0 < c, c' \leq 1$ to preserve Lipschitz condition¹. Assuming extended paths from x_t, x'_t intersect in the original i-ResNet, we can pick c, c' such that $g(x_t) = g(x'_t)$. This preserves the intersection in the new i-ResNet, and makes x_{t+1}, x'_{t+1} equally far from the line $x_t - x'_t$. Now, Theorem 4.5.1 can be applied as above, leading to contradiction; the intersection cannot exist in the original i-ResNet. \square

¹It preserves it for the x_t, x'_t pair. Then, function g that is $\text{Lip}(g) < 1$ over whole \mathbb{R}^p can always be constructed around $g(x_t) = cf(x_t)$ and $g(x'_t) = c'f(x'_t)$ for $0 < c, c' \leq 1$, $\text{Lip}(f) < 1$, e.g. by interpolating g linearly between $g(x_t)$ and $g(x'_t)$ over the segment $[x_t, x'_t]$, replicating the segment endpoints beyond its ends on the line passing through it, and replicating g from the line into the rest of \mathbb{R}^p .

The result allows us to show that i-ResNets faces a similar constraint in its capabilities as Neural ODEs

Theorem 4.5.2. *Let $\mathcal{X} = \mathbb{R}^p$, and let $\mathcal{Z} \subset \mathcal{X}$ and $h : \mathcal{X} \rightarrow \mathcal{X}$ be the same as in Theorem 4.4.1. Then, no p -i-ResNet can model h .*

Proof. Consider a T -layered i-ResNet on \mathcal{X} , giving rise to extended space-time paths in $\mathcal{X} \times [0, T]$, with integer $t \in [0, T]$ corresponding to activations in subsequent layers. For any $x \in \mathcal{Z}$, the extended path in $\mathcal{X} \times [0, T]$ starts at $(x, 0)$ and ends at (x, T) . Since i-ResNet layers are continuous transformations, the union of all extended paths arising from \mathcal{Z} is a simply connected subset of $\mathcal{X} \times [0, T]$; it has no holes and partitions $\mathcal{X} \times [0, T]$ into separate regions. Since extended paths cannot intersect, (x, T) remains in the same region as $(x, 0)$, which is in contradiction with mapping h . \square

The proof shows that the limitation in capabilities of the two architectures for invertible mappings analyzed here arises from the fact that paths in invertible mappings constructed through NeuralODEs and i-ResNets are not allowed to intersect and from continuity in \mathcal{X} .

4.5.2 i-ResNets with Extra Dimensions are Universal Approximators for Homeomorphisms

Similarly to Neural ODEs, expanding the dimensionality of the i-ResNet from p to $2p$ by adding zeros on input guarantees that any p -homeomorphism can be approximated, as long as its Lipschitz constant is finite and an upper bound on it is known during i-ResNet architecture construction.

Theorem 4.5.3. *For any homeomorphism $h : \mathcal{X} \rightarrow \mathcal{X}$, $\mathcal{X} \subset \mathbb{R}^p$ with $\text{Lip}(h) \leq k$, there exists a $2p$ -i-ResNet $\phi : \mathbb{R}^{2p} \rightarrow \mathbb{R}^{2p}$ with $\lfloor k + 4 \rfloor$ residual layers such that $\phi([x, 0^{(p)}]) = [h(x), 0^{(p)}]$ for any $x \in \mathcal{X}$.*

Proof. For a given invertible i-ResNet approximating h , define a possibly non-invertible mapping $\delta(x) = (h(x) - x)/T$, where $T = \lfloor k + 1 \rfloor$; we have $\text{Lip}(\delta(x)) < 1$. An i-ResNet that models h using $T + 3$ layers ϕ_i for $i = 0, \dots, T + 2$ can be constructed in the following way:

$$\begin{aligned} \phi_0([x, 0]) &\rightarrow [x, 0] + [0, \delta(x)], \\ \phi_i([z, y]) &\rightarrow [z, y] + [yT/(T + 1), 0] \quad i = 1, \dots, T + 1, \\ \phi_{T+2}([h(x), \delta(x)]) &\rightarrow [h(x), \delta(x)] + [0, -\delta(x)] \end{aligned}$$

The first layer maps x into $\delta(x)$ and stores it in the second set of p activations. The subsequent $T + 1$ layers progress in a straight line from $[x, \delta(x)]$ to $[h(x), \delta(x)]$ in $T + 1$ constant-length steps, and the last layer restores null values in the second set of p activations.

All layers are continuous mappings. The residual part of the first layer has Lipschitz constant below one, since $\text{Lip}(\delta(x)) < 1$. The middle layers have residual part constant in z and contractive in y , with Lipschitz constant below one. The residual part of the last layer is a mapping of the form $[h, \delta] \rightarrow [0, -\delta]$. For a pair $x, x' \in \mathcal{X}$, let $h = h(x), h' = h(x'), \delta = \delta(x), \delta' = \delta(x')$. We have $\|[0, -\delta] - [0, -\delta']\| = \|\delta - \delta'\| \leq \|[h, \delta] - [h', \delta']\|$, with equality only if $h = h'$. From invertibility of $h(x)$ we have that $h = h'$ implies $x = x'$ and thus $\delta = \delta'$; hence, the residual part of the last layer also has Lipschitz constant below one. \square

The theoretical construction above suggests that while on the order of k layers may be needed to approximate arbitrary homeomorphism $h(x)$ with $\text{Lip}(h) \leq k$, only the first and last layers depend on $h(x)$ and need to be trained; the middle layers are simple, fixed linear layers. The last layer for $x \rightarrow h(x)$ is the same as the first layer of $h(x) \rightarrow x$ would be, but the inverse of the first layer, but since i-ResNet construct invertible mappings $x \rightarrow x + f(x)$ using possibly non-invertible $f(x)$, it has to be trained along with the first layer.

The construction for i-ResNets is similar to that for NeuralODEs, except one does not need to enforce differentiability in the time domain, hence we do not need smooth accumulation and removal of $\delta(x)$ in the second set of p activations, and the movement from x to $h(x)$ in the original p dimensions does not need to be smooth. In both cases, the transition from x to $h(x)$ progresses along a straight line in the first p dimensions, with the direction of movement stored in the second set of p variables.

4.6 Experimental Results

4.6.1 Neural ODEs

We performed experiments to validate if the $q > 2p$ -dimensions threshold beyond which any p -homeomorphism can be approximated by a q -ODE-Net can be observed empirically in a practical classification problem. We used the CIFAR10 dataset [51] that consists of 32×32 RGB images, that is, each input image has a dimensionality of $p = 32 \times 32 \times 3$. We constructed a series of q -ODE-Nets with dimensionality $q \geq p$, and for each measured the cross-entropy loss for the problem of classifying CIFAR10 images into one of ten classes. We used the default split of the dataset into 50,000 training and 10,000 test images.

In designing the architecture of the neural network underlying the ODE, we followed ANODE [41]. Briefly, the network is composed of three 2D convolutional layers. The first two convolutional layers use k filters, and the last one uses the number of input channels as the number of filters, to ensure that the dimensionalities of the input and output of the network match. The convolution stack is followed by a ReLU activation function. A linear layer, with softmax activation and cross-entropy loss, operates on top of the ODE block.

To extended the dimensionality of the space in which the ODE operates, we introduce additional null channels on input; that is, we use input images of form $32 \times 32 \times (3 + d)$. Then, to achieve $q = 2p$, we need $d = 3$; with $d = 4$ we reach $q > 2p$. We tested $d \in \{0, \dots, 7\}$. To analyze how the capacity of the network interplay with the increases in input dimensionality, we also experimented with varying the number of convolutional filters, k , in the layers inside the ODE block. We trained a series of networks with k , increasing from 16 to 128, in increments of 16. We executed all experiments on a single NVIDIA Tesla V100 GPU card, on a machine with 2 Intel Xeon Gold 6146 CPUs and 384 GB RAM, using Python 3.7, PyTorch 1.2, and the tochdiffeq package [23].

The results in Fig. 4.2 show that the rates at which the loss decreases as training progress are influenced by the number of added null input channels. As the input dimensionality reaches past $q = 2p + 1$, that is, once d increases to 4, improvements in convergence rates become smaller. Increasing dimensionality from $d = 0$ to $d = 4$ results in a larger change of convergence rate than increasing d beyond 4. This effect is present irrespective of the size of the network, as captured by the number of convolution filters, k .

Test set loss (Fig. 4.3) also shows diminishing returns from adding dimensions beyond $2p + 1$. For larger networks, with a higher number of convolutional filters, the increased flexibility in routing

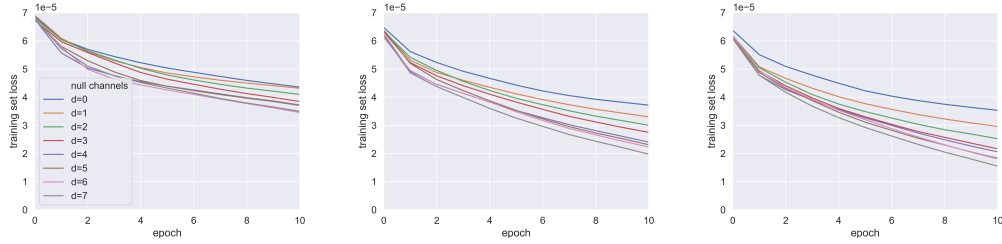


Figure 4.2: Training set cross-entropy loss, for increasing number d of null channels, added to RGB images. For each d , the input images have dimensionality $32 \times 32 \times (3 + d)$. Left: ODE-Net with 16 convolutional filters; center: 64 filters; right: 128 filters.

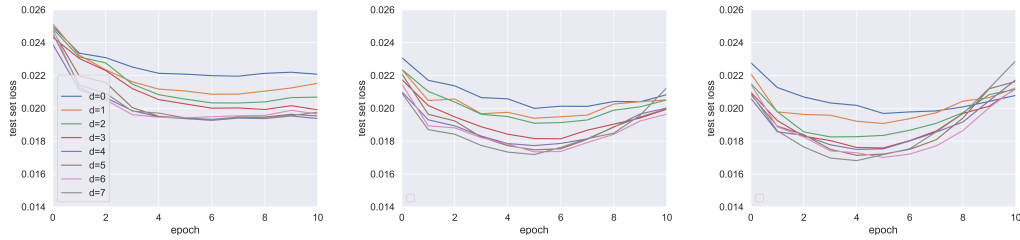


Figure 4.3: Test set cross-entropy loss, for increasing number d of null channels, added to RGB images. For each d , the input images have dimensionality $32 \times 32 \times (3 + d)$. Left: ODE-Net with 16 convolutional filters; center: 64 filters; right: 128 filters.

the ODE trajectories that arises from higher dimensionality leads to lower test set loss early on but eventually results in overtraining.

The change in behavior of the ODE-Net associated with passing over the $2p + 1$ dimensionality threshold becomes even more evident when the minimum of the loss overall epochs is visualized in relation to the input dimensionality, as captured by the number of additional channels, d (Fig. 4.4). For almost every network, irrespective of the number of convolutional filters in the ODE-Net, the minimum loss drops fast as the number of additional channels increases from 0 to 4. Past $d = 4$, that is, beyond $q = 2p + 1$, the decreases in minimum loss, in most cases, slower – both for the training set and the test set loss.

4.6.2 i-ResNets

We tested whether i-ResNet operating in one dimension can learn to perform the $x \rightarrow -x$ mapping, and whether adding one more dimension has impact on the ability learn the mapping. To this end, we constructed a network with five residual blocks. In each block, the residual mapping is a single linear transformation, that is, the residual block is $x_{t+1} = x_t + Wx_t$. We used the official i-ResNet PyTorch package [22] that relies on spectral normalization [52] to limit the Lipschitz constant to less than unity. We trained the network on a set of 10,000 randomly generated values of x uniformly distributed in $[-10, 10]$ for 100 epochs, and used an independent test set of 2,000 samples generated similarly.

For the one-dimensional $x \rightarrow -x$ and the two-dimensional $[x, 0] \rightarrow [-x, 0]$ target mapping, we used MSE as the loss. Adding one extra dimension results in successful learning of the mapping,

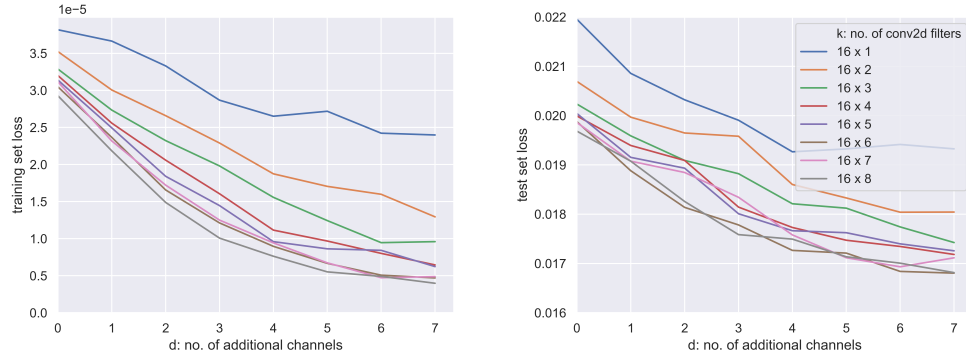


Figure 4.4: Minimum of cross-entropy loss across all epochs as a function of d , the number of null channels added to input images, for ODE-Nets with a different number of convolutional filters; $d = 4$ corresponds to $q \geq 2p + 1$. Left: training set loss; right: test set loss.

confirming Theorem 4.5.3. The test MSE on each output is below 10^{-10} ; the network learned to negate x , and to bring the additional dimension back to null, allowing for invertibility of the model. For the i-ResNet operating in the original, one-dimensional space, learning is not successful (MSE of 33.39), the network learned a mapping $x \rightarrow cx$ for a small positive c , that is, the mapping closest to negation of x that can be achieved while keeping non-intersecting paths, confirming experimentally Corollary 4.5.1.

4.7 Conclusions

Neural ODEs and i-ResNet are recently proposed methods for enforcing invertibility of residual neural models. Having a generic technique for constructing invertible models can open new avenues for advances in learning systems, but so far the question of whether Neural ODEs and i-ResNets can model any continuous invertible function remained unresolved. Here, we showed that both of these models are limited in their approximation capabilities. We then proved how to easily overcome this limitation: any homeomorphism on a p -dimensional Euclidean space can be approximated by a Neural ODE or an i-ResNet operating on an augmented Euclidean space.

Chapter 5

Conclusions and Future Work

In this dissertation proposal, we explored two properties in neural networks: invariance and invertibility. Currently, some network architectures are designed to address certain specific types of invariance – for example, convolutional neural networks internally utilize shift-invariant frequency representation. We consider here the question of learning invariance from data without specifying the type of invariance. In chapter 3, we construct invariant layers with the use of a new, proposed activation function and apply them in the architecture of auto-encoder. We show that the new proposed activation function is better at learning invariant representations in the latent layer of auto-encoder where the encoder part can approximate invariant mapping and the decoder part can approximate the inverse invariant mapping.

The second problem we consider here is invertibility. Typical neural networks are not invertible as information can be lost while it passes through layers of neural networks. Recently proposed Neural ODEs (ODE-Net) and invertible Residual Networks (i-ResNets) give an example of how to generalize residual networks (ResNets) to be invertible. However, ODE-Nets and i-ResNets cannot approximate all types of invertible functions. In chapter 4, we show that with added dimensions on both input and output spaces, Neural ODEs and i-ResNets are capable of approximating any smooth, invertible function.

One example of future work is to join these two lines of research. In the first part of our work, we use the new proposed activation function to learn the invariant representations in the latent layer of auto-encoder, where the decoder part reconstructs the original signal from the latent layer (inverse invariant mapping) and the encoder part can approximate the invariant mapping. In the second part, we have shown that ODE-Nets and i-ResNets can approximate any smooth, invertible function. If we could learn the invariant representation or approximate the invariant mapping by an ODE-Net or i-ResNet in the encoder part with the simple technique showed in chapter 4. Then by the out-of-box invertibility of ODE-Net and i-Resnet, no decoder is needed.

Appendix A

Review on Topology and Manifold Theory

A.1 Topological Spaces

Let's start with topological spaces, which are defined in almost every textbook of topology.

Definition A.1.1. (*Topological Space [53]*). A topology on X is a collection τ of subsets of X having the following properties:

- \emptyset and X are in τ .
- The union of the elements of any subcollection of τ is in τ .
- The intersection of the elements of any finite subcollection of τ is in τ .

We call the sets in τ the open sets, and a topological space is an ordered pair (X, τ) consisting of a set X and a topology τ on X . Still, we often omit specific mention of τ if no confusion arises.

Let X be a topological space, $p \in X$, and $S \subseteq X$ [50].

- A neighborhood of p is an open subset U that $p \in U \subseteq S$.
- S is a closed set if $X \setminus S$ is open
- The interior of S ($\text{Int } S$) is the union of all open subsets of X contained in S .
- The exterior of S ($\text{Ext } S$) is the union of all open subsets of X contained in $X \setminus S$.
- The closure of S (\bar{S}) is the intersection of all closed subsets of X containing S .

Now let's see some examples of topology space.

Definition A.1.2. (*Metric Space [50]*). A metric space is a set X defined with a distance function (also called a metric) $d : X \times X \rightarrow \mathbb{R}$ with the following properties. For all $x, y, z \in X$:

- $d(x, y) \geq 0$; equality holds if and only if $x = y$.
- $d(x, y) = d(y, x)$.
- $d(x, z) \leq d(x, y) + d(y, z)$.

In a metric space X , the open ball with radius $r \geq 0$ at a point $x \in X$ is the set

$$B_r(x) = \{y \in X : d(x, y) < r\}, \quad (\text{A.1})$$

and the closed ball of radius r at a point x is the set $B_r(x) = \{y \in X : d(x, y) \leq r\}$. The metric topology on X is an open subset $S \subseteq M$ (for every point $x \in S$, there is some $r > 0$ such that $B_r(x) \subseteq S$). If X is a metric space and S is any subset of X , the restriction of metric d to pairs of points in S turns S into a metric space.

Definition A.1.3. (Euclidean Space [50]). An n -dimensional Euclidean space is the set \mathbb{R}^n of ordered n -tuples of real numbers with integer $n \geq 1$. A point in \mathbb{R}^n is written as (x^1, \dots, x^n) , (x^i) , or x ; each x^i is called the coordinate of x . When $n = 0$, \mathbb{R}^0 is the one-element set 0 .

The Euclidean norm of defined on each $x \in \mathbb{R}^n$ is

$$\|x\| = \sqrt{(x^1)^2 + \dots + (x^n)^2}, \quad (\text{A.2})$$

with $|X| \geq 0$ and it forms a metric. The Euclidean distance function on points $x, y \in \mathbb{R}^n$ is

$$d(x, y) = \|x - y\|. \quad (\text{A.3})$$

This metric topology that turns \mathbb{R}^n into a metric space is called the Euclidean topology, and the subsets of Euclidean spaces are also metric spaces.

Example A.1.1. Let X is any set. A discrete topology on X is the collection of all subsets of X and the trivial topology is the topology only consists of X and \emptyset .

We can define various spaces in topological spaces. As the idea of the second part of this paper implements manifold theory, if we define an arbitrary topological space, it could be too general with some unwanted properties. For example, the X from the trivial topology (Example A.1.1) does not have sufficiently many open subsets. Thus we add the following restriction on our topological spaces.

Definition A.1.4. (Hausdorff Space [50]). A topological space X is a Hausdorff space if for every pair $p, q \in X$, with $p \neq q$, there exist disjoint open subsets $U, V \subseteq X$ such that $p \in U$ and $q \in V$.

Continuous mapping and homeomorphism are really important in this paper. Let X and Y be topological spaces.

Definition A.1.5. (Continuous Mappings [50]). A map $F : X \rightarrow Y$ is said to be continuous if for every open subset $U \subseteq Y$, the preimage $F^{-1}(U)$ is open in X .

Definition A.1.6. (Open and Closed Map [50]) A map $F : X \rightarrow Y$ (continuous or not) is said to be an open map if for every open subset $U \subseteq X$, the image set $F(U)$ is open in Y , and a closed map if for every closed subset $K \subseteq X$, the image $F(K)$ is closed in Y . Continuous maps may be open, closed, both, or neither, as can be seen by examining simple examples involving subsets of the plane.

Definition A.1.7. (Homeomorphism [50]). A continuous bijection map $F : X \rightarrow Y$ with continuous inverse is called a homeomorphism. If there exists a homeomorphism from X to Y , we say that X and Y are homeomorphic.

Definition A.1.8. (Local Homeomorphism [50]). A continuous map $F : X \rightarrow Y$ is said to be a local homeomorphism if every point $p \in X$ has a neighborhood $U \subseteq X$ such that $F(U)$ is open in Y and F restricts to a homeomorphism from U to $F(U)$.

A.2 Bases, Countability, and Compactness

Definition A.2.1. (Base). The basis \mathcal{B} of the topology on the topological space X is a collection of open subsets of X . Generally speaking, if X is a set, the basis \mathcal{B} has the properties of $X = \cup_{B \in \mathcal{B}} B$ and if $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 \cap B_2$, then there exists $B_3 \in \mathcal{B}$ such that $x \in B_3 \subseteq B_1 \cap B_2$.

Definition A.2.2. (Countability). A countable set is finite or countable infinite. A topological space X is first-countable when there is a countable neighborhood basis at every point in X , and second-countable when there is a countable basis for its topology. As a countable basis for X contains a countable neighborhood basis at each point, second-countability implies first-countability.

Before we define compactness of a set, let's see what open cover is.

Definition A.2.3. (Open Cover [53]). A collection of A of subsets of a space X is said to cover X , or to be a covering of X , if the union of the elements of A is equal to X . It is called an open covering of X if its elements are open subsets of X .

Definition A.2.4. (Compactness [53]). A space X is said to be compact if every open covering A of X contains a finite subcollection that also covers X .

A.3 Subspaces, Products and Disjoint Unions

In this section, we introduce several ways of building new topological spaces. We will omit the properties of product topology and disjoint union topology as they follow the property of subspace with minor differences.

Definition A.3.1. (Subspaces [50]). Let Y be a non-empty subset of a topological space X . The collection $\tau_Y = \{O \cap Y : O \in X\}$ of subsets of Y is said to be a subspace topology (or the relative topology or the induced topology) on Y or the topology induced on Y by τ , and then, the topological space (Y, τ_Y) is said to be a subspace of (X, τ) .

Proposition A.3.1. (Properties of the Subspace Topology [50]). Let X be a topological space and let S be a subspace of X .

- CHARACTERISTIC PROPERTY: If Y is a topological space, a map $F : Y \rightarrow S$ is continuous if and only if the composition $\iota_S \circ F : Y \rightarrow X$ is continuous, where $\iota_S : S \hookrightarrow X$ is the inclusion map (the restriction of the identity map of X to S).
- the subspace topology is the unique topology on S for which the characteristic property holds.
- A subset $K \subseteq S$ is closed in S if and only if there exists a closed subset $L \subseteq X$ such that $K = L \cap S$.
- The inclusion map $\iota_S : S \hookrightarrow X$ is a topological embedding.
- If Y is a topological space and $F : X \rightarrow Y$ is continuous, then $F|_S : S \rightarrow Y$ (the restriction of F to S is continuous.)
- If \mathcal{B} is a basis for the topology of X , then $\mathcal{B}_S = \{B \cap S : B \in \mathcal{B}\}$ is a basis for the subspace topology on S .
- If X is Hausdorff, then so is S .
- If X is first-countable, then so is S .
- If X is second-countable, then so is S .

Definition A.3.2. (Product Spaces [54]). The Cartesian product of finitely many sets X_1, \dots, X_k is an ordered k -tuple with the form of $X_1 \times \dots \times X_k$. The i th projection map is $\pi_i : X_1 \times \dots \times X_k \rightarrow X_i$ with $\pi_i(x_1, \dots, x_k) = x_i$. Suppose X_1, \dots, X_k are topological spaces. The collection of all subsets of $X_1 \times \dots \times X_k$ has the form of $U_1 \times \dots \times U_k$, where each U_i is open in X_i , forms a basis for a topology on $X_1 \times \dots \times X_k$, and this collection is called the product topology. The production space with the production topology is called a product space.

Taking disjoint unions of other space is another simple way of building new topological spaces.

Definition A.3.3. (Disjoint Union Spaces [54]). If $(X_\alpha)_{\alpha \in A}$ is an indexed family of sets, their disjoint union is the set

$$\coprod_{\alpha \in A} X_\alpha = \{(x, \alpha) : \alpha \in A, x \in X_\alpha\} \quad (\text{A.4})$$

There is a canonical injective map for each α , $\iota_\alpha : X_\alpha \rightarrow \coprod_{\alpha \in A} X_\alpha$ give by $\iota_\alpha(x) = (x, \alpha)$, and the images of these maps for different values of α are disjoint.

A.4 Quotient Spaces

We need quotient mapping and equivalence relation to define quotient spaces.

Definition A.4.1. (Quotient Mapping [54]). Let (X, τ) and (Y, τ_1) be topological spaces. A surjective mapping $\pi : (X, \tau) \rightarrow (Y, \tau_1)$ with the following property

$$\text{For each subset } U \text{ of } Y, U \in \tau_1 \iff \pi^{-1}(U) \in \tau. \quad (\text{A.5})$$

is said to be a quotient mapping.

Theorem A.4.1. (Properties of Quotient Maps [50]). Let $\pi : X \rightarrow Y$ be a quotient map.

- CHARACTERISTIC PROPERTY: If B is a topological space, a map $F : Y \rightarrow B$ is continuous if and only if $F \circ \pi : X \rightarrow B$ is continuous.
- The quotient topology is the unique topology on Y for which the characteristic property holds.
- A subset $K \subseteq Y$ is closed if and only if $\pi^{-1}(K)$ is closed in X .
- If π is injective then it is a homeomorphism.
- If $U \subseteq X$ is a saturated open or closed subset, then the restriction $\pi|_U : \rightarrow \pi(U)$ is a quotient map.
- And composition of π with another quotient map is again a quotient map.

Definition A.4.2. (Equivalence Relation [54]). A binary relation \sim on a set X is said to be an equivalence relation if it is reflexive, symmetric and transitive; that is, for all $a, b, c \in X$,

- $a \sim a$ (reflexive);
- $a \sim b \implies b \sim a$ (symmetric);
- $a \sim b$ and $b \sim c \implies a \sim c$ (transitive).

If $a, b \in X$, then a and b are said to be in the same equivalence class if $a \sim b$.

Definition A.4.3. (Quotient Spaces [50]). Suppose X is a topological space and \sim is an equivalence relation on X . Let X/\sim denote the set of equivalence classes and let $\pi : X \rightarrow X/\sim$ be the quotient map sending each point in X to its equivalence class. Endowed with the quotient topology determined by π , the space X/\sim is called the quotient space (or identification space) of X determined by \sim .

The two theorems below on quotient maps play important roles in topology and smooth manifolds.

Definition A.4.4. (Fiber). A fiber of a map $f : X \rightarrow Y$ is the preimage of an element $y \in Y$. That is,

$$f^{-1}(y) = \{x \in X : f(x) = y\}. \quad (\text{A.6})$$

Theorem A.4.2. (Passing to the Quotient [50]). Suppose $\pi : X \rightarrow Y$ is a quotient map, B is a topological space, and $F : X \rightarrow B$ is a continuous map that is constant on the fibers of π (i.e., $\pi(p) = \pi(q)$ implies $F(p) = F(q)$). Then there exists a unique continuous map $\bar{F} : Y \rightarrow B$ such that $F = \bar{F} \circ \pi$.

Theorem A.4.3. (Uniqueness of Quotient Spaces [50]). If $\pi_1 : X \rightarrow Y_1$ and $\pi_2 : X \rightarrow Y_2$ are quotient maps that are constant on each other's fibers (i.e., $\pi_1(p) = \pi_1(q)$ if and only if $\pi_2(p) = \pi_2(q)$), then there exists a unique homeomorphism $\varphi : Y_1 \rightarrow Y_2$ such that $\varphi \circ \pi_1 = \pi_2$.

Now let's see some examples of quotient spaces.

Example A.4.1. (Cone [54]). For any space (X, τ) , the cone (CX, τ_1) over X is the quotient space $((X, \tau) \times I)/\sim$, where I denotes the unit interval with its usual topology and \sim is the equivalence relation $(x, 0) \sim (x', 0)$, for all $x, x' \in X$.

Example A.4.2. (Suspension [54]). For any topological space (X, τ) the suspension, (SX, τ_2) , is the quotient space of the cone (CX, τ_1) obtained by identifying the points $(x, 1) \sim (x', 1)$. Thus (SX, τ_2) is the quotient space of $(X, \tau) \times I$ where the equivalence relation is

$$(x, 1) \sim (x', 1) \text{ and } (x, 0) \sim (x', 0), \text{ for all } x, x' \in X : \quad (\text{A.7})$$

A.5 Smooth Manifolds

A.5.1 Topological Manifolds

Definition A.5.1. (Topological Manifold [50]). Suppose M is a topological space. We say that M is a topological manifold of dimension n or a topological n -manifold if it has the following properties:

- Hausdorff space: for every pair of distinct points $p, q \in M$, there are disjoint open subsets $\mathcal{U}, \mathcal{V} \subset M$ such that $p \in \mathcal{U}$ and $q \in \mathcal{V}$.
- Second-countable: there exists a countable basis for the topology of M .
- Locally Euclidean of dimension n : each point of M has a neighborhood that is homeomorphic to an open subset of \mathbb{R}^n . More specifically, for each $p \in M$ we can find
 - an open subset $\mathcal{U} \subset M$ containing p ,
 - an open subset $\hat{\mathcal{U}} \subset \mathbb{R}^n$, and
 - a homeomorphism $\varphi : \mathcal{U} \rightarrow \hat{\mathcal{U}}$.

The definition of topological manifold does not consider spaces of mixed dimensions.

Theorem A.5.1. (Topological Invariance of Dimension [50]). *A nonempty n -dimensional topological manifold cannot be homeomorphic to an m -dimensional manifold unless $m = n$.*

Topological manifolds are typically formed by specifying coordinate charts.

Definition A.5.2. (Coordinate Chart [50]). *Let M be a topological n -manifold. A coordinate chart (or just a chart) on M is a pair (U, φ) , where U is an open subset of M and $\varphi : U \rightarrow \tilde{U}$ is a homeomorphism from U to an open subset $\tilde{U} = \varphi(U) \subseteq \mathbb{R}^n$.*

The set U is called a coordinate domain or a coordinate neighborhood of each of its points. If $\varphi(U)$ is an open ball in \mathbb{R}^n , then U is called a coordinate ball; if $\varphi(U)$ is an open cube, U is a coordinate cube. The map φ is called a (local) coordinate map, and the component functions (x^1, \dots, x^n) of φ , defined by $\varphi(p) = (x^1(p), \dots, x^n(p))$, are called local coordinates on U .

Now, let's see some examples of topological manifolds.

Example A.5.1. (Spheres [50]). *For each integer $n \geq 0$, the unit n -sphere \mathbb{S}^n is Hausdorff and second-countable because it is a topological subspace of \mathbb{R}^{n+1} . To show that it is locally Euclidean, for each index $i = 1, \dots, n+1$ let U_i^+ denote the subset of \mathbb{R}^{n+1} where the i th coordinate is positive:*

$$U_i^+ = \{(x^1, \dots, x^{n+1}) \in \mathbb{R}^{n+1} : x^i > 0\} \quad (\text{A.8})$$

Similarly, U_i^- is the set where $x^i < 0$. Let $f : \mathbb{B}^n \rightarrow \mathbb{R}$ be the continuous function

$$f(u) = \sqrt{1 - |u|^2}. \quad (\text{A.9})$$

Then for each $i = 1, \dots, n+1$, it is easy to check that $U_i^+ \cap \mathbb{S}^n$ is the graph of the function

$$x^i = f(x^1, \dots, \hat{x}^i, \dots, x^{n+1}), \quad (\text{A.10})$$

where the hat indicates that x^i is omitted. Similarly, $U_i^- \cap \mathbb{S}^n$ is the graph of

$$x^i = -f(x^1, \dots, \hat{x}^i, \dots, x^{n+1}), \quad (\text{A.11})$$

Thus, each subset $U_i^\pm \cap \mathbb{S}^n$ is locally Euclidean of dimension n , and the maps $\varphi_i^\pm : U_i^\pm \cap \mathbb{S}^n \rightarrow \mathbb{B}^n$ given by

$$\varphi_i^\pm(x^1, \dots, x^{n+1}) = (x^1, \dots, \hat{x}^i, \dots, x^{n+1}) \quad (\text{A.12})$$

are graph coordinates for \mathbb{S}^n . Since each point of \mathbb{S}^n is in the domain of at least one of these $2n+2$ charts, \mathbb{S}^n is a topological n -manifold.

A.5.2 Smooth Structures

In the preceding section, we have introduced the basic definitions related to topological manifolds. There is also a good reason to mention calculus as we try to add derivatives to the functions or maps on a topological manifold, and such derivatives cannot be invariant under homeomorphisms. Thus we need some new structures of the topology on a manifold, and the manifold allows us to decide which functions to or from the manifold are smooth. Let's first review the smooth maps between Euclidean spaces.

Definition A.5.3. (Smooth Function and Diffeomorphism in Euclidean Spaces [50]). If U and V are open subsets of Euclidean spaces \mathbb{R}^n and \mathbb{R}^m , respectively, a function $F : U \rightarrow V$ is said to be smooth (or C^∞ , or infinitely differentiable) if it has continuous partial derivatives of all orders. If in addition F is bijective and has a smooth inverse map, it is called a diffeomorphism. A diffeomorphism is always a homeomorphism.

We need to consider collecting all smooth charts to form a new structure on M do define the smoothness on a manifold. Let's first define transition maps.

Definition A.5.4. (Transition Map [50]) Let M be a topological n -manifold. If $(U, \varphi), (V, \psi)$ are two charts such that $U \cap V \neq \emptyset$, the composite map $\psi \circ \varphi^{-1} : \varphi(U \cap V) \rightarrow \psi(U \cap V)$ is called the transition map from φ to ψ . It is a composition of homeomorphisms and is therefore, itself a homeomorphism.

Definition A.5.5. (Smooth Compatibility [50]). Two charts (U, φ) and (V, ψ) are said to be smoothly compatible if either $U \cap V = \emptyset$ or the transition map $\psi \circ \varphi^{-1}$ is a diffeomorphism.

Since $\varphi(U, \varphi)$ and $\psi(V, \psi)$ are open subsets of \mathbb{R}^n , and according to Definition A.5.3, smoothness of this map can be interpreted in the sense of having continuous derivatives of any orders.

Definition A.5.6. (Atlas [50]). An atlas for M is a collection of charts whose domains cover M . An atlas \mathcal{A} is called a smooth atlas if any two charts in \mathcal{A} are smoothly compatible with each other.

We can show that an atlas is smooth by verifying that each transition map $\psi \circ \varphi^{-1}$ is smooth whenever (U, φ) and (V, ψ) are charts in \mathcal{A} ; once we have proved this, we have the diffeomorphism $\psi \circ \varphi^{-1}$ as its inverse $(\psi \circ \varphi^{-1})^{-1} = \varphi \circ \psi^{-1}$ is already a smooth transition map. The ultimate plan is to define a "smooth structure" on M by giving a smooth atlas, and a smooth function $f : M \rightarrow \mathbb{R}$ with $f \circ \varphi^{-1}$ is smooth for each coordinate chart (U, φ) in the atlas. Now we can define the concept of smooth structure, followed by the definition of smooth manifold.

Definition A.5.7. (Smooth Structure [50]). If M is a topological manifold, a smooth structure on M is a maximal smooth atlas which takes the union of all atlases in a smooth structure.

Definition A.5.8. (Smooth Manifold). A smooth manifold is a pair (M, \mathcal{A}) , where M is a topological manifold and \mathcal{A} is a smooth structure on M . Smooth structures are also called differentiable structures or C^∞ structures. We also use the term smooth manifold structure to mean a manifold topology together with a smooth structure.

Generally, it is very inconvenient to define a smooth structure by explicitly expressing a maximal smooth atlas, because there could be many charts contained in such atlas. Fortunately, the next proposition shows that we only need to specify some smooth atlas.

Proposition A.5.1. ([50]). Let M be a topological manifold.

- Every smooth atlas \mathcal{A} for M is contained in a unique maximal smooth atlas, called the smooth structure determined by \mathcal{A} .
- Two smooth atlases for M determine the same smooth structure if and only if their union is a smooth atlas.

Let us see two examples of smooth manifolds before we proceed into the general theory.

Example A.5.2. (Euclidean Spaces [50]). For each non-negative integer n , the Euclidean space \mathbb{R}^n is a smooth n -manifold with the smooth structure determined by the atlas consisting of the single chart $(\mathbb{R}^n, Id_{\mathbb{R}^n})$. We call this the standard smooth structure on \mathbb{R}^n and the resulting coordinate map standard coordinates. Unless we explicitly specify otherwise, we always use this smooth structure on \mathbb{R}^n . With respect to this smooth structure, the smooth coordinate charts for \mathbb{R}^n are exactly those charts (U, φ) such that φ is a diffeomorphism (in the sense of ordinary calculus) from U to another open subset $\hat{U} \subseteq \mathbb{R}^n$.

Example A.5.3. (Spheres [50]). We showed in section A.1 that the n -sphere $\mathbb{S}^n \subseteq \mathbb{R}^{n+1}$ is a topological n -manifold. We put a smooth structure on \mathbb{S}^n as follows. For each $i, \dots, n+1$, let (U_i^\pm, φ_i^\pm) denote the graph coordinate charts we constructed in section 4.1.2.1. For any distinct indices i and j , the transition map $\varphi_i^\pm \circ (\varphi_j^\pm)^{-1}$ is easily computed. In the case $i < j$, we get

$$\varphi_i^\pm \circ (\varphi_j^\pm)^{-1}(u^1, \dots, u^n) = (u^1, \dots, \hat{U}u^i, \dots, \pm\sqrt{1 - |u|^2}, \dots, u^n) \quad (\text{A.13})$$

(with the square root in the j th position), and similar formula holds when $i > j$. When $i = j$, an even simpler computation gives $\varphi_i^+ \circ (\varphi_i^-)^{-1} = \varphi_i^- \circ (\varphi_i^+)^{-1} = Id_{\mathbb{R}^n}$. Thus, the collection of charts $\{(U_i^\pm, \varphi_i^\pm)\}$ is a smooth atlas, and so defines a smooth structure on \mathbb{S}^n . We call this its standard smooth structure.

Example A.5.4. (Smooth Product Manifolds [50]). If M_1, \dots, M_k are smooth manifolds of dimensions n_1, \dots, n_k , respectively, we showed in section 4.1.2.2 that the product space $M_1 \times \dots \times M_n$ is a topological manifold of dimension $n_1 + \dots + n_k$, with charts of the form $(U_1 \times \dots \times U_k, \varphi_1 \times \dots \times \varphi_k)$. Any two such charts are smoothly compatible because, as is easily verified,

$$(\psi_1 \times \dots \times \psi_k) \circ (\varphi_1 \times \dots \times \varphi_k)^{-1} = (\psi_1 \circ \varphi_1^{-1}) \times \dots \times (\psi_k \circ \varphi_k^{-1}), \quad (\text{A.14})$$

which is a smooth map. This defines a natural smooth manifold structure on the product, called the product smooth manifold structure. For example, this yields a smooth manifold structure on the n -torus $\mathbb{T}^n = \mathbb{S}^1 \times \dots \times \mathbb{S}^1$.

In each of the previous examples, the smooth structure is constructed in two steps: we first check if the topological space is a topological manifold or not, and then we specify a smooth structure.

A.5.3 Manifolds with Boundary

In many applications of manifolds, we encounter spaces that are smooth manifolds except that they have some "boundary." For example, the closed balls in \mathbb{R}^n and closed hemispheres in \mathbb{S}^n . Before we extend the definition of manifolds for such spaces, let's introduce some notations.

Definition A.5.9. (Closed n -dimensional Upper Half-Space [50]). Points in topological manifolds with boundary will have neighborhoods modeled either on open subsets of \mathbb{R}^n or on open subsets of the closed n -dimensional upper half-space $\mathbb{H}^n \subseteq \mathbb{R}^n$, defined as

$$\mathbb{R}^n = \{(x^1, \dots, x^n) \in \mathbb{R}^n : x^n \geq 0\}. \quad (\text{A.15})$$

Int \mathbb{H}^n and $\partial \mathbb{H}^n$ to denote the interior and boundary of \mathbb{H}^n , respectively, as a subset of \mathbb{R}^n . When $n > 0$, this means

$$\text{Int } \mathbb{H}^n = \{(x^1, \dots, x^n) \in \mathbb{R}^n : x^n > 0\}, \quad \partial \mathbb{H}^n = \{(x^1, \dots, x^n) \in \mathbb{R}^n : x^n = 0\}. \quad (\text{A.16})$$

In the $n = 0$ case, $\mathbb{H}^0 = \mathbb{R}^0 = \{0\}$, so $\text{Int } \mathbb{H}^0 = \mathbb{R}^0$ and $\partial \mathbb{H}^0 = \emptyset$.

Definition A.5.10. (Topological Manifold With Boundary [50]). An n -dimensional topological manifold with boundary is a second-countable Hausdorff space M in which every point has a neighborhood homeomorphic either to an open subset of \mathbb{R}^n or to a (relatively) open subset of \mathbb{H}^n .

Theorem A.5.2. (Topological Invariance of the Boundary). If M is a topological manifold with boundary, then each point of M is either a boundary point or an interior point, but not both. Thus ∂M and $\text{Int}M$ are disjoint sets whose union is M .

We use redundant phrases even though manifold with boundary contains manifold. We use manifold without boundary to refer the original manifold in previous sections and manifold with or without boundary to emphasize that the boundary might be empty.

A.5.4 Smooth Maps

Function and map are terms that technically synonymous, but it is often better to make a slight distinction between them in studying smooth manifolds. In this paper, we refer a function for a map whose codomain in a real-valued function or a vector-valued function. Map or mapping can be any maps, such as a map between manifolds.

Definition A.5.11. (Smooth Functions [50]). Suppose M is a smooth n -manifold, k is a non-negative integer, and $f : M \rightarrow \mathbb{R}^k$ is any function. We say that f is a smooth function if for every $p \in M$ whose domain contains p and such that the composite function $f \circ \varphi^{-1}$ is smooth on the open subset $\hat{U} = \varphi(U) \subseteq \mathbb{R}^n$

Definition A.5.12. (Coordinate Representation [50]). Given a function $f : M \rightarrow \mathbb{R}^k$ and a chart (U, φ) for M , the function $\hat{f} : \varphi(U) \rightarrow \mathbb{R}^k$ denoted by $\hat{f}(x) = f \circ \varphi^{-1}(x)$ is called the coordinate representation of f .

The maps between manifold are generalized smooth functions.

Definition A.5.13. (Smooth Maps Between Manifolds [50]). Let M, N be smooth manifolds, and let $F : M \rightarrow N$ be any map. We say that F is a smooth map if for every $p \in M$, there exist smooth charts (U, φ) containing p and (V, ψ) containing $F(p)$ such that $F(U) \subseteq V$ and the composite map $\psi \circ F \circ \varphi^{-1}$ is smooth from $\varphi(U)$ to $\psi(V)$

Proposition A.5.2. ([50]). Every smooth map is continuous.

Definition A.5.14. (Diffeomorphism [50]). If M and N are manifolds with or without boundary, a diffeomorphism from M to N is a smooth bijective map $F : M \rightarrow N$ that has a smooth inverse. We say that M and N are diffeomorphic if there exists a diffeomorphism between them. Sometimes this is symbolized by $M \approx N$.

Proposition A.5.3. (Properties of Diffeomorphisms).

- Every composition of diffeomorphisms is a diffeomorphism
- Every finite product of diffeomorphisms between smooth manifolds is a diffeomorphism.
- Every diffeomorphism is a homeomorphism and an open map.
- The restriction of a diffeomorphism to an open submanifold with or without boundary is a diffeomorphism onto its image.
- "Diffeomorphic" is an equivalence relation on the class of all smooth manifolds with or without boundary.

A.6 Embeddings

A.6.1 Maps of Constant Rank

Definition A.6.1. (*Rank [50]*). Suppose M and N are smooth manifolds with or without boundary. Given a smooth map $F : M \rightarrow N$ and a point $p \in M$, we define the rank of F at p to be the rank of the linear map $dF_p : T_p M \rightarrow T_{F(p)} N$; it is the dimension of $\text{Im } dF_p \subseteq T_{F(p)} N$. If F has the same rank r at every point, we say that it has constant rank, and write $\text{rank } F = r$. Because the rank of a linear map is never higher than the dimension of either its domain or its codomain, the rank of F at each point is bounded above by the minimum of $\{\dim M, \dim N\}$. If the rank of dF_p is equal to this upper bound, we say that F has full rank at p , and if F has full rank everywhere, we say F has full rank.

The most important constant-rank maps are those of full rank.

Proposition A.6.1. (*Submersion and Immersion [50]*). Suppose $F : M \rightarrow N$ is a smooth map and $p \in M$. If dF_p is surjective, then p has a neighborhood U such that $F|_U$ is a submersion. If dF_p is injective, then p has a neighborhood U such that $F|_U$ is an immersion.

Smooth submersions and immersions are also locally surjective and injective, respectively.

Definition A.6.2. (*Local Diffeomorphism [50]*). If M and N are smooth manifolds with or without boundary, a map $F : M \rightarrow N$ is called a local diffeomorphism if every point $p \in M$ has a neighborhood U such that $F(U)$ is open in N and $F|_U : U \rightarrow F(U)$ is a diffeomorphism. The next theorem is the key to the most important properties of local diffeomorphisms.

Proposition A.6.2. (*Properties of Local Diffeomorphisms [50]*).

- Every composition of local diffeomorphisms is a local diffeomorphism.
- Every finite product of local diffeomorphisms between smooth manifolds is a local diffeomorphism.
- Every local diffeomorphism is a local homeomorphism and an open map.
- The restriction of a local diffeomorphism to an open submanifold with or without boundary is a local diffeomorphism.
- Every diffeomorphism is a local diffeomorphism.
- Every bijective local diffeomorphism is a diffeomorphism.
- A map between smooth manifolds with or without boundary is a local diffeomorphism if and only if in a neighborhood of each point of its domain, it has a coordinate representation that is a local diffeomorphism.

Proposition A.6.3. (*[50]*). Suppose M and N are smooth manifolds (without boundary), and $F : M \rightarrow N$ is a map.

- F is a local diffeomorphism if and only if it is both a smooth immersion and a smooth submersion.
- $\dim M = \dim N$ and F is either a smooth immersion or a smooth submersion, then it is a local diffeomorphism.

A.6.2 Smooth Embeddings

Embedding is a special kind of immersion, and it's particularly essential.

Definition A.6.3. (*Smooth Embedding [50]*). If M and N are smooth manifolds with or without boundary, a smooth embedding of M into N is a smooth immersion $F : M \rightarrow N$ that is also a topological embedding. A smooth embedding is a map that is both a topological embedding and a smooth immersion, not just a topological embedding that happens to be smooth.

The next proposition shows some simple sufficient criteria for when injective immersion is an embedding.

Proposition A.6.4. (*[50]*). Suppose M and N are smooth manifolds with or without boundary, and $F : M \rightarrow N$ is an injective smooth immersion. If any of the following holds, then F is a smooth embedding.

- F is an open or closed map.
- F is a proper map.
- M is compact.
- M has empty boundary and $\dim M = \dim N$.

Theorem A.6.1. (*Local Embedding Theorem [50]*). Suppose M and N are smooth manifolds with or without boundary, and $F : M \rightarrow N$ is a smooth map. Then F is a smooth immersion if and only if every point in M has a neighborhood $U \subseteq M$ such that $F|_U : U \rightarrow N$ is a smooth embedding.

A.7 Submanifolds

A.7.1 Embedded Submanifolds

Definition A.7.1. (*Embedded Submanifold [50]*). Suppose M is a smooth manifold with or without boundary. An embedded submanifold of M is a subset $S \subseteq M$ that is a manifold (without boundary) in the subspace topology, endowed with a smooth structure with respect to which the inclusion map $S \hookrightarrow M$ is a smooth embedding. Embedded submanifolds are also called regular submanifolds by some authors.

Definition A.7.2. (*Dimension of Embedded Submanifold [50]*). If S is an embedded submanifold of M , the difference $\dim M - \dim S$ is called the codimension of S in M , and the containing manifold is called the ambient manifold for S . An embedded hypersurface is an embedded submanifold of codimension 1. The empty set is an embedded submanifold of any dimension.

The following proposition shows several ways of producing embedded submanifolds.

Proposition A.7.1. (*Images of Embeddings as Submanifolds [50]*). Suppose M is a smooth manifold with or without boundary, N is a smooth manifold, and $F : N \rightarrow M$ is a smooth embedding. Let $S = F(N)$. With the subspace topology, S is a topological manifold, and it has a unique smooth structure making it into an embedded submanifold of M with the property that F is diffeomorphism onto its image.

The previous proposition means that the images of smooth embeddings are the embedded submanifolds.

Proposition A.7.2. (Slices of Product Manifolds [50]). Suppose M and N are smooth manifolds. For each $p \in N$, the subset $M \times \{p\}$ (called a slice of the product manifold) is an embedded submanifold of $M \times N$ diffeomorphic to M

Let's see two examples.

Example A.7.1. (Product Manifolds [50]). Suppose M_1, \dots, M_k are topological manifolds of dimensions n_1, \dots, n_k , respectively. The product space $M_1 \times \dots \times M_k$ is shown to be a topological manifold of dimension $n_1 + \dots + n_k$ as follows. It is Hausdorff and second-countable by the Propositions A.7.1 and A.7.2, so only the locally Euclidean property needs to be checked. Given any point $(p_1, \dots, p_k) \in M_1 \times \dots \times M_k$, we can choose a coordinate chart (U_i, φ_i) for each M_i with $p_i \in U_i$. The product map

$$\varphi_1 \times \dots \times \varphi_k : U_1 \times \dots \times U_k \rightarrow \mathbb{R}^{n_1 + \dots + n_k} \quad (\text{A.17})$$

is a homeomorphism onto its image, which is a product open subset of $\mathbb{R}^{n_1 + \dots + n_k}$. Thus, $M_1 \times \dots \times M_k$ is a topological manifold of dimension $n_1 + \dots + n_k$, with charts of the form $(U_1 \times \dots \times U_k, \varphi_1 \times \dots \times \varphi_k)$. Any two such charts are smoothly compatible

$$(\psi_1 \times \dots \times \psi_k) \circ (\varphi_1 \times \dots \times \varphi_k)^{-1} = (\psi_1 \circ \varphi_1^{-1}) \times \dots \times (\psi_k \circ \varphi_k^{-1}), \quad (\text{A.18})$$

which is a smooth map. This defines a smooth manifold structure on the product.

Example A.7.2. (Tori [50]). For a positive integer n , the n -torus is the product space $\mathbb{T}^n = \mathbb{S}^1 \times \dots \times \mathbb{S}^1$. By the discussion above, it is a smooth n -manifold. (The 2-torus is usually called simply the torus.)

We have to define proper map as merely being an embedded submanifold is not quite a strong enough condition.

Definition A.7.3. (Proper Map [50]). An embedded submanifold $S \subseteq M$ is said to be properly embedded if the inclusion $S \hookrightarrow M$ is a proper map.

Proposition A.7.3. ([50]) Suppose M is a smooth manifold with or without boundary and $S \subseteq M$ is an embedded submanifold. Then S is properly embedded if and only if it is a closed subset of M .

The next theorem shows that embedded submanifolds are modeled locally of \mathbb{R}^k into \mathbb{R}^n , identifying \mathbb{R}^k with the subspace

$$\{(x^1, \dots, x^k, x^{k+1}, \dots, x^n) : x^{k+1} = \dots = x^n = 0\} \subseteq \mathbb{R}^n \quad (\text{A.19})$$

The definition below generalizes the previous statement.

Definition A.7.4. (k -slice [50]). If U is an open subset of \mathbb{R}^n and $k \in \{0, \dots, n\}$, a k -dimensional slice of U (or simply a k -slice) is any subset of the form

$$S = \{(x^1, \dots, x^k, x^{k+1}, \dots, x^n) \in U : x^{k+1} = c^{k+1}, \dots, x^n = c^n\} \quad (\text{A.20})$$

for some constants c^{k+1}, \dots, c^n . (When $k = n$, this just means $S = U$.) Clearly, every k -slice is homeomorphic to an open subset of \mathbb{R}^k .

Theorem A.7.1. (Local Slice Criterion for Embedded Submanifolds). Let M be a smooth n -manifold. If $S \subseteq M$ is an embedded k -dimensional submanifold, then S satisfies the local k -slice condition. Conversely, if $S \subseteq M$ is a subset that satisfies the local k -slice condition, then with the subspace topology, S is a topological manifold of dimension k , and it has a smooth structure making it into a k -dimensional embedded submanifold of M .

A.7.2 Restricting Maps to Submanifolds

When the domain or codomain of a smooth map $F : M \rightarrow N$ is restricted to a submanifold, the smoothness of the restricted map is important to know.

Theorem A.7.2. (Restricting the Domain of a Smooth Map [50]). *If M and N are smooth manifolds with or without boundary, $F : M \rightarrow N$ is a smooth map, and $S \subseteq M$ is an immersed or embedded submanifold, then $F|_S : S \rightarrow N$ is smooth.*

The resulting map is not guaranteed to be smooth when the codomain of a smooth map is restricted. The next theorem includes sufficient conditions for a map to be smooth when its codomain is restricted.

Theorem A.7.3. (Restricting the Codomain of a Smooth Map). *Suppose M is a smooth manifold (without boundary), $S \subseteq M$ is an immersed submanifold, and $F : N \rightarrow M$ is a smooth map whose image is contained in S . If F is continuous as a map from N to S , then $F : N \rightarrow S$ is smooth.*

When the submanifold S is embedded, the hypothesis of continuity always holds.

Corollary A.7.1. (Embedded Case [50]). *Let M be a smooth manifold and $S \subseteq M$ be an embedded submanifold. Then every smooth map $F : N \rightarrow M$ whose image is contained in S is also smooth as a map from N to S .*

A.8 The Whitney Theorems

A.8.1 The Whitney Embedding Theorem

In this section, we show that every smooth n -manifold with or without boundary is diffeomorphic to a properly embedded submanifold (with or without boundary) of \mathbb{R}^{2n+1} (the main idea of Whitney Embedding Theorem [50]) by steps.

Theorem A.8.1. (Whitney Embedding Theorem [50]). *Every smooth n -manifold with or without boundary admits a proper smooth embedding into \mathbb{R}^{2n+1} .*

Corollary A.8.1. ([50]). *Every smooth n -dimensional manifold with or without boundary is diffeomorphic to a properly embedded submanifold (with or without boundary) of \mathbb{R}^{2n+1} .*

Theorem A.8.2. (Strong Whitney Embedding Theorem [50]). *If $n > 0$, every smooth n -manifold admits a smooth embedding into \mathbb{R}^{2n} .*

A.8.2 The Whitney Approximation Theorems

Definition A.8.1. (δ -close [50]). *If $\delta : M \rightarrow \mathbb{R}$ is a positive continuous function, we say that two functions $F, \tilde{F} : M \rightarrow \mathbb{R}^k$ are δ -close if $|F(x) - \tilde{F}(x)| < \delta(x)$ for all $x \in M$.*

Theorem A.8.3. (Whitney Approximation Theorem for Functions [50]). *Suppose M is a smooth manifold with or without boundary, and $F : M \rightarrow \mathbb{R}^k$ is a continuous function. Given any positive continuous function $\delta : M \rightarrow \mathbb{R}$, there exists a smooth function $\tilde{F} : M \rightarrow \mathbb{R}^k$ that is δ -close to F . If F is smooth on a closed subset $A \subseteq M$, then \tilde{F} can be chosen to be equal to F on A .*

Corollary A.8.2. ([50]). *If M is a smooth manifold with or without boundary and $\delta : M \rightarrow \mathbb{R}$ is a positive continuous function, there is a smooth function $e : M \rightarrow \mathbb{R}$ such that $0 < e(x) < \delta(x)$ for all $x \in M$.*

Theorem A.8.4. (**Whitney Approximation Theorem** [50]). *Suppose N is a smooth manifold with or without boundary, M is a smooth manifold (without boundary), and $F : N \rightarrow M$ is a continuous map. Then F is homotopic to a smooth map. If F is already smooth on a closed subset $A \subseteq N$, then the homotopy can be taken to be relative to A .*

Corollary A.8.3. (**Extension Lemma for Smooth Maps** [50]). *Suppose N is a smooth manifold with or without boundary, M is a smooth manifold, $A \subseteq N$ is a closed subset, and $f : A \rightarrow M$ is a smooth map. Then f has a smooth extension to N if and only if it has a continuous extension to N .*

A.9 Flows

In this section, we recapitulate the material in [47, 48, 55].

Definition A.9.1. (**Topological Transformation Group or Flow**). *A topological transformation group or a flow is an ordered triple $(\mathcal{X}, \mathcal{G}, \Phi)$ involving an additive group \mathcal{G} with neutral element 0, and a mapping $\Phi : \mathcal{X} \times \mathcal{G} \rightarrow \mathcal{X}$ such that $\Phi(x, 0) = x$ and $\Phi(\Phi(x, s), t) = \Phi(x, s + t)$ for all $x \in \mathcal{X}$, all $s, t \in \mathcal{G}$. Further, mapping $\Phi(x, t)$ is assumed to be continuous with respect to the first argument. The mapping Φ gives rise to a parametric family of homeomorphisms $\phi_t : \mathcal{X} \rightarrow \mathcal{X}$ defined as $\phi_t(x) = \Phi(x, t)$, with the inverse being $\phi_t^{-1} = \phi_{-t}$.*

Definition A.9.2. (**Orbit or Trajectory**). *Given a flow, an orbit or a trajectory associated with $x \in \mathcal{X}$ is a subspace $G(x) = \{\Phi(x, t) : t \in \mathcal{G}\}$. Given $x, y \in \mathcal{X}$, either $G(x) = G(y)$ or $G(x) \cap G(y) = \emptyset$; two orbits are either identical or disjoint, they never intersect. A point $x \in \mathcal{X}$ is a fixed point if $G(x) = \{x\}$.*

Definition A.9.3. (**Discrete Flow**). *A discrete flow is defined by setting $\mathcal{G} = \mathcal{Z}$. For arbitrary homeomorphism h of \mathcal{X} onto itself, we easily get a corresponding discrete flow, an iterated discrete dynamical system, $\phi_0(x) = x$, $\phi_{t+1} = h(\phi_t(x))$, $\phi_{t-1}(x) = h^{-1}(\phi_t(x))$.*

Definition A.9.4. (**Continuous Flow**). *A type of flow relevant to Neural ODEs is a continuous flow, defined by setting $\mathcal{G} = \mathcal{R}$, and adding an assumption that the family of homeomorphisms, the function $\Phi : \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{X}$, is differentiable with respect to its second argument, t , with continuous $d\Phi/dt$. The key difference compared to a discrete flow is that the flow at time t , $\phi_t(x)$, is now defined for arbitrary $t \in \mathcal{R}$, not just for integers. We will use the term p -flow to indicate that $\mathcal{X} \subset \mathcal{R}^p$.*

Informally, in a continuous flow, the orbits are continuous, and the property that orbits never intersect has consequences for what homeomorphisms ϕ_t can result from a flow. Unlike in the discrete case, for a given homeomorphism h there may not be a continuous flow such that $\phi_T = h$ for some T . We cannot just set $\phi_T = h$, what is required is a continuous family of homeomorphisms ϕ_t such that $\phi_T = h$ and ϕ_0 is identity – such family may not exist for some h . In such a case, a Neural ODE would not be able to model the mapping h .

Appendix B

List of Abbreviations

Table B.1: List of Abbreviations

Symbol	Description
ML	Machine learning
3D	Three-dimensional object
GPU	Graphics processing unit
MLP	Multi-layer perceptron
CNN	Convolutional neural network
RNN	Recurrent neural network
AD	Automatic differentiation
MSE	Mean squared error
SGD	Stochastic gradient descent
ResNet	Residual network
i-ResNet	Invertible residual network
ODE	Ordinary differential equations
IVP	Initial value problem
FFT	Fast Fourier transform, an algorithm that computes the discrete Fourier transform (DFT) of a sequence

Appendix C

List of Notations

C.1 Notations in Chapter 2

Table C.1: Notations in Chapter 2: Fig. 2.1

Symbol	Description
I_1, I_2, I_3, I_4	Input variables
v_1 and v_2	Intermediate variables
o	Output variable
$\bar{I}_1, \bar{I}_2, \bar{I}_3, \bar{I}_4$	Adjoint for I_1, I_2, I_3 and I_4
\bar{v}_1, \bar{v}_2	Adjoint for v_1 and v_2
\bar{o}	Adjoint for o

Table C.2: Notations in Chapter 2: Sizes and Dataset

Symbol	Description
m	Trainig set size (number of examples)
p	Input size (number of features)
c	Output size (number of classes)
n	Number of internal parameters (size of θ , the weight in the neural network)
L and l_i	L : the total number of layers in a neural network; l_i is the number of neurons in the i -th layer; $i \in \{1, 2, \dots, L\}$; The 0-th layer indicates the input layer
X	The set of training examples (each column is an example)
$\{x^{(1)}, \dots, x^{(m)}\}$	A batch of m training examples randomly selected from X ; Superscript $x^{(i)}$ is used to denote the i -th sample in this batch.
x and x_j	A random example (column vector) from X ; x_j : the j -th component of x
y and y_j	y : the target associated with x ; y_j : j -th element of y
\hat{y} and \hat{y}_j	\hat{y} is the predicted y ; \hat{y}_j : the j -th component of \hat{y}
$\{y^{(1)}, \dots, y^{(m)}\}$	Set of targets associated with $\{x^{(1)}, \dots, x^{(i)}, \dots, x^{(m)}\}$; Superscript $y^{(i)}$ is used to denote the target associate with $x^{(i)}$; $\hat{y}^{(m)}$ is the predicted $y^{(m)}$

Table C.3: Notations in Chapter 2 and Chapter 3: Neural Network

Symbol	Description
w_i	The weight for the i -th layer
$w_{i,j,k}$	Weight for the k -th input of the j -th neuron in the i -th layer
b_i	The bias for the i -th layer
$b_{i,j}$	The bias for the input of the j -th neuron in the i -th layer
a_i	The output for the i -th layer
$a_{i,j}$	The output for the j -th neuron in the i -th layer
g_i	The activation for the i -th layer
f	A neural network that maps set of training examples $\{x^{(1)}, \dots, x^{(m)}\}$ to the corresponding set of targets $\{y^{(1)}, \dots, y^{(m)}\}$

C.2 Notations in Chapter 3

Table C.4: Notations in Chapter 3

Symbol	Description
f and g	f : encoder; g : decoder
(x, z)	The input dataset consists of pairs of (x, z) ; x : input vector; z : latent representation or code
y	The output vector, re-constructed x
f and g	f : encoder (mapping x to z with $z = f(x)$); g : decoder (mapping z to y)
$z = FFT(x)$	Python function: <code>numpy.fft</code> (compute the one-dimensional discrete Fourier Transform)

C.3 Notations in Chapter 4

Table C.5: Notations in Chapter 4: Dataset

Symbol	Description
$\mathcal{X} \subset \mathbb{R}^p$	Input space
x	Input vector $x \in \mathcal{X}$
$[T] = \{1, \dots, T\}$	A sequence of time
$x[i]$	The i -th component of vector x
Superscript $x^{(p)}$	The dimensionality of vectors; that is, $0^{(p)} \in \mathbb{R}^{(p)}$
x_0 and x_t	x_0 is initial input vector and x_t is input vector at time t , where $t \in [T]$

Table C.6: Notations in Chapter 4: Neural ODEs

Symbol	Description
θ	Weight vector
F	A neural network block (a function that maps input vector to output vector required to be almost everywhere differentiable with respect to both of its arguments)
$F(x, \theta) \in \mathbb{R}^{p'}$	Output vector.
$F(x, \theta) = x + f(x, \theta)$	A residual block; The input and output of a residual block have the same dimensionality p ; f is some differentiable, nonlinear, possibly multi-layer
$x_{t+1} = x_t + f_t(x_t, \theta_t)$	Residual blocks are usually stacked in a sequence, and the functional form of f_t is often the same for all blocks $t \in [T]$ in the sequence
$x_{t+1} - x_t = f_{\Theta}(x_t, t)$	Re-write the sequence of residual blocks; Θ consists of trainable parameters for all blocks in the sequence; t is used to pick the proper subset of parameters, θ_t
$\phi_T(x_0) = x_T$	The result of applying a sequence of T residual blocks to the initial input x_0
f_{Θ}	If arbitrary f_{Θ} is allowed, the sequence of residual blocks can, in principle, model arbitrary mappings $x \rightarrow \phi_T(x_0)$

Bibliography

- [1] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
- [2] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [3] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 1969.
- [4] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.
- [5] Stuart Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.
- [6] David E Rumelhart and Geoffrey E Hintonf. Learning representations by back-propagating errors. *Nature*, 323:9, 1986.
- [7] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.
- [8] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [9] John J Weng, Narendra Ahuja, and Thomas S Huang. Learning recognition and segmentation of 3-d objects from 2-d images. In *1993 (4th) International Conference on Computer Vision*, pages 121–128, 1993.
- [10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
- [11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.
- [12] Xu Shen, Xinmei Tian, Anfeng He, Shaoyan Sun, and Dacheng Tao. Transform-invariant convolutional neural networks for image classification and search. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1345–1354, 2016.

- [13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
- [15] Ian Goodfellow, Honglak Lee, Quoc V Le, Andrew Saxe, and Andrew Y Ng. Measuring invariances in deep networks. In *Advances in neural information processing systems*, pages 646–654, 2009.
- [16] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.
- [17] Kihyuk Sohn and Honglak Lee. Learning invariant representations with local transformations. *arXiv preprint arXiv:1206.6418*, 2012.
- [18] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 215–223, 2011.
- [19] Koray Kavukcuoglu, Pierre Sermanet, Y-Lan Boureau, Karol Gregor, Michaël Mathieu, and Yann L Cun. Learning convolutional feature hierarchies for visual recognition. In *Advances in neural information processing systems*, pages 1090–1098, 2010.
- [20] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on computer vision and pattern recognition*, pages 2528–2535. IEEE, 2010.
- [21] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.
- [22] Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *International Conference on Machine Learning*, pages 573–582, 2019.
- [23] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.
- [24] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153), 2018.
- [25] Charles C Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1305, 2019.
- [26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

- [27] Jian Huang Lai, Pong C Yuen, and Guo Can Feng. Face recognition using holistic fourier invariant features. *Pattern Recognition*, 34(1):95–109, 2001.
- [28] Alireza Khotanzad and Yaw Hua Hong. Invariant image recognition by zernike moments. *IEEE Transactions on pattern analysis and machine intelligence*, 12(5):489–497, 1990.
- [29] Tomasz Arodz. Invariant object recognition using radon-based transform. *Computing and Informatics*, 24(2):183–199, 2012.
- [30] Bernard Haasdonk and Hans Burkhardt. Invariant kernel functions for pattern analysis and machine learning. *Machine learning*, 68(1):35–61, 2007.
- [31] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 991–999, 2015.
- [32] Dmitry Laptev, Nikolay Savinov, Joachim M Buhmann, and Marc Pollefeys. Ti-pooling: transformation-invariant pooling for feature learning in convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 289–297, 2016.
- [33] Gong Cheng, Junwei Han, Peicheng Zhou, and Dong Xu. Learning rotation-invariant and fisher discriminative convolutional neural networks for object detection. *IEEE Transactions on Image Processing*, 28(1):265–278, 2018.
- [34] Gustavo Deco and Wilfried Brauer. Nonlinear higher-order statistical decorrelation by volume-conserving neural architectures. *Neural Networks*, 8(4):525–535, 1995.
- [35] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538, 2015.
- [36] Lev Semenovich Pontryagin, EF Mishchenko, VG Boltyanskii, and RV Gamkrelidze. *The mathematical theory of optimal processes*. Wiley, 1962.
- [37] Chris Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. DiffEqFlux.jl-a Julia library for neural differential equations. *arXiv preprint arXiv:1902.02376*, 2019.
- [38] Amir Gholami, Kurt Keutzer, and George Biros. ANODE: unconditionally accurate memory-efficient gradients for neural ODEs. *arXiv preprint arXiv:1902.10298*, 2019.
- [39] Tianjun Zhang, Zhewei Yao, Amir Gholami, Kurt Keutzer, Joseph Gonzalez, George Biros, and Michael Mahoney. ANODEV2: A coupled neural ODE evolution framework. *arXiv preprint arXiv:1906.04596*, 2019.
- [40] Tian Qi Chen, Jens Behrmann, David K Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. In *Advances in Neural Information Processing Systems*, pages 9913–9923, 2019.
- [41] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural ODEs. *arXiv preprint arXiv:1904.01681*, 2019.

- [42] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [43] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.
- [44] Hongzhou Lin and Stefanie Jegelka. Resnet with one-neuron hidden layers is a universal approximator. In *Advances in neural information processing systems*, pages 6169–6178, 2018.
- [45] Marion Kirkland Fort. The embedding of homeomorphisms in flows. *Proceedings of the American Mathematical Society*, 6(6):960–967, 1955.
- [46] Stephen A Andrea. On homeomorphisms of the plane, and their embedding in flows. *Bulletin of the American Mathematical Society*, 71(2):381–383, 1965.
- [47] WR Utz. The embedding of homeomorphisms in continuous flows. In *Topology Proc*, volume 6, pages 159–177, 1981.
- [48] Michael Brin and Garrett Stuck. *Introduction to dynamical systems*. Cambridge university press, 2002.
- [49] William Browder. Manifolds with $\pi_1 = z$. *Bulletin of the American Mathematical Society*, 72(2):238–244, 1966.
- [50] John M Lee. *Introduction to smooth manifolds*. Springer, 2001.
- [51] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [52] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.
- [53] James Munkres. *Topology*. Pearson Education, 2014.
- [54] Sidney A Morris. *Topology without tears*. University of New England, 1989.
- [55] Laurent Younes. *Shapes and diffeomorphisms*, volume 171. Springer, 2010.

Vita

Han Zhang was born on November 5, 1990, in Dandong, China. He joined the department of Statistical Sciences and Operation Research at Virginia Commonwealth University in 2011. He received his Bachelor of Science in Mathematical Sciences with a concentration in statistics in 2013. He received a Master of Science in Mathematical Sciences from Virginia Commonwealth University in 2015. On the year, he received his certification in Base Programmer for SAS 9. He joined the department of Computer Science doctoral program in 2015.