Adaptive Robust Pooling and Feature Projections in Deep Declarative Networks

Wenbo Du

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honours)
The Australian National University

June 2021

Except where otherwise indicated, this thesis is my own original work.

Wenbo Du
9 June 2021

To my mother, she is my hero.

Acknowledgments

Last four year has been an incredible journey. I came to study in Australia in April 2017 and started my degree in engineering at the Australian National University. Later I transferred to computer science because I found it more enjoyable. It wasn't easy initially, but I overcame many adversaries and became a better version of myself.

This thesis would not be possible without the collaboration and help from Stephen Gould. Stephen is one of the most knowledgeable researchers in computer vision and machine learning. It was a great honour to pursue my research directions under his supervision. His creativity and passion have profoundly influenced and shaped the way I think about academic problems. Importantly, Stephen is extremely patient: he is willing to discuss research questions that may seem trivial at his level of knowledge; he is willing to explain the same research ideas many times.

I would like to thank Ramesh Sankaranarayana and Miaomiao Liu for being my thesis examiners. Although I have not got opportunities to work with them, I knew them impressive research accomplishments in multiple fields.

Furthermore, I would like to thank Dylan Campbell for providing me with so many valuable research suggestions during the group meeting. His brilliant ideas gave me tremendous help last semester. I appreciate the assistance from other people on my honours project, including Tanya Dixit, Cui Ruikai, Robert Jeffrey, Chinmay Garg, Minrui Zhao, Yuchong Yao, Wenjun Yang, Suikei Wang, and Itzik ben Shabat.

I want to express my appreciation to many people and organisations. The Australian National University provides me with a friendly and collaborative academic atmosphere. I had a great exchange semester at McGill University. Gleb Belov, Daniel Harabor, Guansong Pang, Paul Scott, Phil Kilby and Hilbert van Pelt kindly gave me many opportunities.

On the personal side, I would like to thank my friends Dipen Thomas, Ziqi Zhang, Xinrui Wei and Weijie Tu for providing me with valuable advice. My friend Zhengzi Xiang back in China always encourages me. Music from Swedish artist Avicii empowers me during the hardest time. Most importantly, my mother, Xuemei Bai, gives me unconditional support. I hope to make her proud.

Abstract

Embedding mathematic optimisation problems as network layers have given exceptional effectiveness in various applications. Deep declarative networks (DDN) generalise multiple previous works in this area: the forward passing of DDN involves solving an optimisation problem. And the backward passing employs the implicit function theorem to obtain gradients. Because DDN can co-exist with conventional network nodes, we can subsume specific layers of traditional neural networks with DDN layers to tackle particular problems.

Robust pooling is an instance of DDN. Prior work indicated robust pooling is more robust to outlier than average and max pooling. This project appends more flexibility to robust pooling by substituting the constant parameter with a learnable one. As an extension, we adapt an innovative loss function as pooling. Moreover, L_p sphere and ball projections are common operations in deep learning. DDN facilitates us to embed L_1, L_2 and L_∞ ball and sphere projections as network layers. In this project, we introduce gradient-based learning for the parameters in the projection layers.

In experiments, we show our adaptive robust pooling delivers significant performance improvement in point cloud classification. We demonstrate our adaptive feature projections outperform non-adaptive ones consistently in image classification. Importantly, our approaches exceed previous methods with negligible extra memory and processing time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimisation problems aim to determine the best solutions among all feasible ones. They have been widely applied in numerous fields, including finance (portfolio optimisation), transport scheduling and data fitting [Boyd and Vandenberghe, 2004].

Recently, multiple state-of-the-art works bridge neural networks and optimisation problems: OptNet [Amos and Kolter, 2017] integrates quadratic programs as network layers. SATNet [Wang et al., 2019] blends logic reasoning problems as differentiable MAXSAT layers into deep learning. Neural ODE [Chen et al., 2018] parameterises the derivative of the hidden states in neural networks. These models are capable of tackling issues that are identified as hard for regular networks.

Deep declarative networks (DDN) [Gould et al., 2020] conclude many previous works on differential optimisation in neural networks. The significant benefit of DDN is its nodes can co-exist with conventional networks. The presence of DDN nodes in networks does not obstacle the parameter updating of other layers.

Similar to conventional networks, DDN has parametric and non-parametric nodes. In non-parametric settings, we are only interested in the gradient of input. In parametric settings, we additionally require gradients of parameters. Theoretically, the gradient computation for DDN input and parameters both rely on the implicit function theorem.

In application aspects, current work mainly concerns non-parametric DDN. This project explores the parametric variations of two DDN instances. We will:

1. Review robust pooling and feature projections as DDN instances and introduce their adaptive versions.
2. Resolve some problems that do not emerge in non-parametric DDN.
3. Evaluate our adaptive versions on point cloud classification and image classification.

1.1 Outlines

We begin with the background and related work in Chapter 2. In Section 2.1, we review fundamental concepts of convolutional neural networks. In particular, we focus on pooling layers. We then explore an innovative loss function that generalises

multiple existing loss functions in Section 2.2. Last, in Section 2.3, we study deep declarative networks, the theoretical foundation for the project. We will talk about how these topics are related to our project.

In Chapter 3, we first review the concept of robust pooling. We then present our deviations and purpose adaptive robust pooling in Section 3.1. Moreover, we formulate the innovative loss function introduced as a pooling layer in Section 3.2. We use DDN theory to compute the gradients of our pooling layers in Section 3.3. To finish, we conduct experiments on point cloud classification and image classification tasks in Section 3.4 and Section 3.5.

In Chapter 4, we cover some existing work on feature projections. Next, we propose our adaptive versions of feature projections in Section 4.1 and derive their gradients in Section 4.2. In Section 4.3, we then investigate adaptive feature projections with image classification tasks. Finally, we summarise the project and identify possible future work in Chapter 5.

1.2 Contributions

We introduced adaptive robust pooling and feature projections as parametric DDN instances. Specifically, we formulated a loss function from [Barron, 2019] as a pooling layer. For all purposed methods, we gave their gradients formulas using DDN rules. In experiments, we showed our adaptive methods outperform their non-adaptive versions with little extra cost.

From the implementation perspective, we wrote several tutorials in jupyter notebooks for demonstration. We gave Python PyTorch implementation of adaptive robust pooling and feature projections as modularised DDN layers. We provided code to reproduce experiments. Our implementation is well-documented and easy to apply in other circumstances. All code and instructions can be found at: `https://github.com/WenboDu1228/ddn_pooling_and_projections`

Chapter 2

Background and Related Work

This chapter first revisits the concept of convolutional neural networks (CNN) in Section 2.1. In Section 2.1.1 and Section 2.1.2, we discuss pooling layers in CNN and the learning process in general neural networks. In Section 2.2, we raise the difficulties with outlier data in machine learning and how researchers handle it. We then introduce a state-of-the-art loss function generalising many existing loss functions in Section 2.2.1. We also present some special cases of this function. Furthermore, Section 2.2.2 analyses the function's probability density distribution and gradient-based learning process. We review some applications and explain how they are related to this project in Section 2.2.3.

We cover deep declarative networks (DDN) from [Gould et al., 2020] in Section 2.3. We first explain the related background and theoretical framework. We then give the overview structure of DDN in Section 2.3.1, including forward passing and backward propagation. In particular, we identify its similarities and differences with conventional neural networks. Moreover, we consider three common DDN cases and their learning process in Section 2.3.2. Specifically, we show their gradient computations. Last, we briefly discuss the applications of DDN and how it builds the frameworks of this project in Section 2.3.4.

2.1 Convolutional Neural Networks

Convolutional neural networks (CNN) are dominant in computer vision tasks [Yamashita et al., 2018]. CNN outperforms traditional neural networks because it captures image spiral dependency. In general, CNN has three parts:

1. **Convolutional layers** are responsible for feature learning from the input. It composed of multiple kernels defined by width and height. The kernels are stacked and contain hyper-parameters such as stride and padding. Convolutional layers utilise parameter sharing to reduce parameters.

2. **Pooling layers** perform non-linear down-sampling. They extract informative features from convolutional layers. After the operations, a smaller set of features is extracted from the original features.

3. **Fully connected layers** are normally placed at the end of CNN models. They take learned features from convolutional layers as input and output prediction scores.

In our project, all experiments in Chapter 3 and Chapter 4 use CNN architectures. Figure 2.1 is an example of CNN model for image classification.

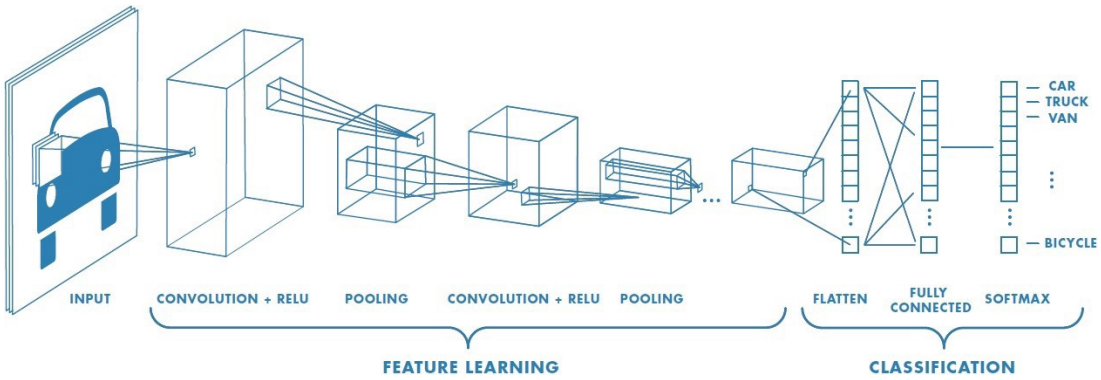

Figure 2.1: CNN for image classification [Saha, 2018]

2.1.1 Pooling Layers

Pooling is one of the most prevailing downsampling techniques. In general, a pooling layer is followed by the applications of rectified linear unit (ReLU) or other non-linear activation functions. Classifying by algorithms, there are two types of pooling used in deep learning:

- **Average pooling**, also called mean pooling, calculates the mean of each patch of features as the output. The output of average pooling can be interpreted as the summary of all features.
- **Max pooling** determines the max values for each patch of features as the output. It serves as a denoising method along with the pooling dimension. Recently, studies have shown that max pooling outperforms average pooling [Yu et al., 2014; Zhou et al., 2016a].

Figure 2.2 are examples of average pooling and max pooling.

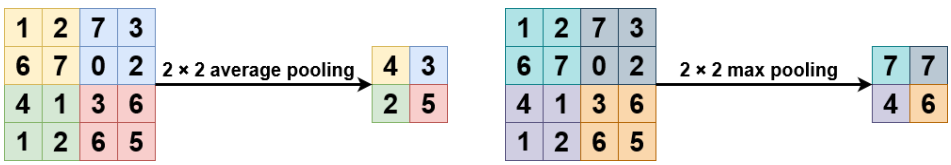

Figure 2.2: **Left:** average pooling. **Right:** max pooling.

We can also distinguish pooling by operation regions:

- **Local pooling** downsamples each patch of features into a single value. It is the most frequently-used pooling method in deep learning [Zhou et al., 2016a]. Our examples in Figure 2.2 are local pooling. Both instances perform pooling over four patches of features.

- **Global pooling** summarises all features into a single value. It is more robust to spatial transformation and hence helps to reduce overfitting [Lin et al., 2014]. Global pooling has shown excellent results on object localization [Zhou et al., 2016a]. Popular models such as ResNet [He et al., 2016], DenseNet [Huang et al., 2017], AlexNet [Iandola et al., 2017] and GoogLeNet [Szegedy et al., 2015] have global average pooling at the end of network architectures. Figure 2.3 shows how global average pooling (GAP) operates on 3-channel input features.

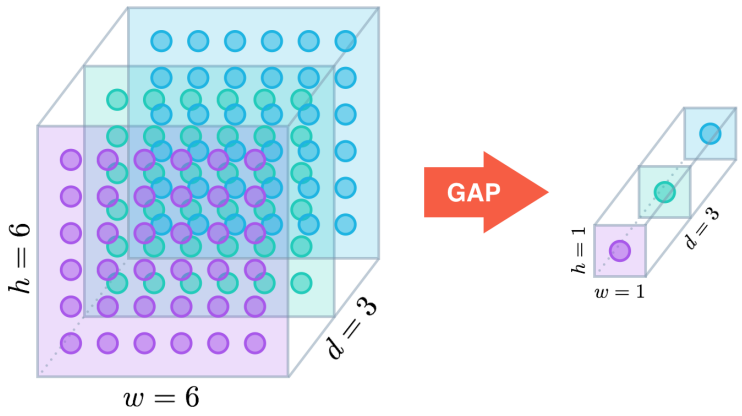

Figure 2.3: Global average pooling [Cook, 2017].

In Section 3.4 and Section 3.5, we substitute global max pooling and global average pooling in two distinct networks with our adaptive robust pooling.

2.1.2 Learning in Neural Networks

Artificial neural networks learn via backpropagation, which calculates the gradients of the loss function for input and network parameters. The iterative parameter updating from one network layer to others relies on the chain rule.

For layers that do not contain learnable parameters (non-parametric layers), we only compute the gradient of input. For instance, rectified linear unit (ReLU) is a popular non-parametric layer [Xu et al., 2015]. For layers with learnable parameters (parametric layers), end-to-end learning requires gradients for layer parameters. Batchnorm [Ioffe and Szegedy, 2015] is a common parametric layer: it helps with training speed and reducing overfitting by ensuring networks have zero mean and

standard deviation. The algorithm is defined as:

$$\begin{aligned} u_B &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \delta_B^2 &= \frac{1}{m}\sum_{i=1}^{m}(x_i - u_B)^2 \\ \bar{x}_i &= \frac{x_i}{\sqrt{\delta_B^2 + \xi}} \\ y_i &= \bar{x}_i + \beta \end{aligned} \tag{2.1}$$

where β and ξ are both trainable parameters.

Our innovative network layers in Chapter 3 are classified as parametric network layers: they are instances of DDN, and therefore the gradient calculation is different from conventional network layers. However, the learning process discussed in this section still applies. We will discuss the learning procedures of DDN layers in Section 2.3.

2.2 A General and Adaptive Robust Loss Function

Data that lies at an abnormal distance from the majority are called outliers [Maddala, 1992]. Outliers may be caused by multiple sources such as data corruption, measurement or input error [Grubbs, 1969]. The exact definition of outliers varies according to the applications. Figure 2.4 is an instance where outliers are added to linearly sampled two-dimensional data.

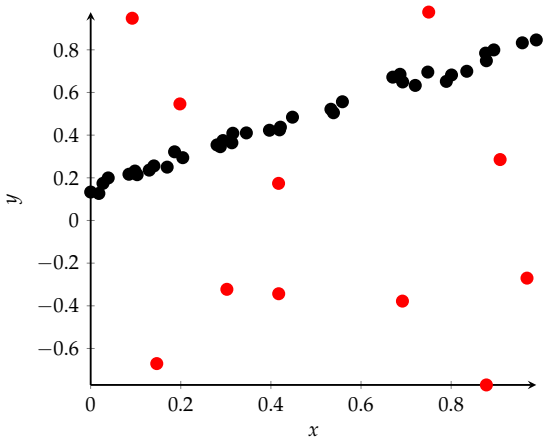

Figure 2.4: Data sampled with linear regression model (block dots) and some outliers (red dots).

In optimisation and statistics, many problems demand models less influenced by outliers. Such property is called **robustness**. [Huber, 1981; Hastie et al., 2015]. In some tasks, researchers favour robust loss functions because they are less sensitive to outlier data. Some popular choices of robust loss functions include Huber loss [Huber, 1964], pseudo-Huber loss [Charbonnier et al., 1997], Welsch loss [Dennis and Welsch, 1978; Leclerc, 1989] and truncated quadratic loss [Gould et al., 2020]. As these functions are closely related to the deviation of robust pooling, we will cover

them in Section 3.1.

2.2.1 Definition

Most robust penalty functions are hand-tuned and appropriate for particular data distributions. Researchers may need experiments with different robust loss functions for data with unknown distribution, which is computationally expensive and challenging for transferring to other applications. For instance, one may train on clean images but perform the test on contaminated images.

The general and adaptive robust loss function introduced by [Barron, 2019] is a generalisation of many known loss functions:

$$f(x, \alpha, c) = \frac{|\alpha-2|}{\alpha}\left(\left(\frac{(\frac{x}{c})^2}{|\alpha-2|}+1\right)^{\frac{\alpha}{2}}-1\right) \tag{2.2}$$

where α is the robustness parameter that defines how outliers are being penalised, and c is the outlier threshold that determines if input x is inlier or outlier. [Barron, 2019] summarises special cases with different α:

$$f(x,\alpha,c) = \begin{cases} \frac{1}{2}(\frac{x}{c})^2 & \text{if } \alpha \to 2 \text{ (squared loss)} \\ \sqrt{(\frac{x}{c})^2+1}-1 & \text{if } \alpha = 1 \text{ (pseudo-Huber loss)} \\ log(\frac{1}{2}(\frac{x}{c})^2+1) & \text{if } \alpha \to 0 \text{ (Cauchy loss, [Black and Ananda, 1996]} \\ 1-exp(-\frac{1}{2}(\frac{\alpha}{c})^2) & \text{if } \alpha \to -\infty \text{ (Welsch loss)} \\ f(x,\alpha,c) = \frac{|\alpha-2|}{\alpha}\left(\left(\frac{(\frac{x}{c})^2}{|\alpha-2|}+1\right)^{\frac{\alpha}{2}}-1\right) & \text{otherwise} \end{cases} \tag{2.3}$$

where we take limit for $\alpha = 2, 0$ and $-\infty$. Figure 2.5 illustrates the graph of Equation (2.2) with some special α values. We can see the all functions overlap when $x \in [-c, c]$ as this range defines inliers. When $x \in (-\infty, c) \cup (c, \infty)$, functions start to have different shapes: if $\alpha = 2$, Equation (2.2) simulates squared loss and has same penalties for all $x \in \mathbb{R}$. For other α values, Equation (2.2) becomes more flat for $x \in (-\infty, c) \cup (c, \infty)$, which indicates less penalties for outliers.

2.2.2 Gradient-based Optimisation

Equation (2.2) is smooth for x, α and c, and hence it is suitable for gradient-based optimisation over input and parameters. We can automatically adjust and tune the parameters for different data distribution [Geman and McClure, 1985]. To enable gradient-based learning, [Barron, 2019] acquires the derivative respect to input x of

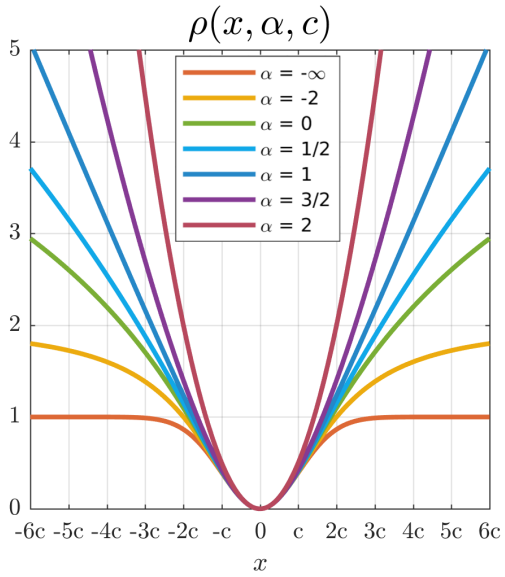

Figure 2.5: The general and adptive robust loss function (Equation (2.2)) with different α [Barron, 2019].

Equation (2.3) as follow:

$$\mathrm{D}_x f(x, \alpha, c) = \begin{cases} \frac{x}{c^2} & \text{if } \alpha = 2 \text{ (squared loss)} \\ \frac{x}{c\sqrt{\frac{x^2}{c} + 1}} & \text{if } \alpha = 1 \text{ (pseudo-Huber loss)} \\ \frac{2x}{x^2 + 2c^2} & \text{if } \alpha = 0 \text{ (Cauchy loss, [Black and Ananda, 1996])} \\ \frac{x}{c^2} exp(-\frac{1}{2}(\frac{x}{c})^2) & \text{if } \alpha = -\infty \text{ (Welsch loss)} \\ \frac{x}{c^2}(\frac{(\frac{x}{c})^2}{|\alpha - 2|} + 1)^{\frac{\alpha}{2} - 1} & \text{otherwise.} \end{cases} \tag{2.4}$$

Figure 2.6 shows the derivative of Equation (2.2) for x with different α. We see the slopes of all gradients overlaps for in inlier range $x \in [-c, c]$. The gradient for squared loss ($\alpha = 2$) keeps linear. For all other α values, the graph slopes vary for $x \in (-\infty, c) \cup (c, \infty)$.

As discovered in [Barron, 2019], optimisers can minimise Equation (2.2) by taking α as small as possible. After enabling the learning of α and c, optimisers will actively cheat on the loss function. To resolve this issue, [Barron, 2019] forms Equation (2.1) as a probability density function:

$$p(x|\mu, \alpha, c) = \frac{1}{cZ(\alpha)} exp(-\phi(x - \mu, \alpha, c)) \tag{2.5}$$

where $Z(\alpha)$ is the integration and approximated by the modified Bessel function of the second kind. Equation (2.5) is a superset of Gaussian distribution ($\alpha = 2$), Cauchy distribution ($\alpha = 0$) and generalized Gaussian distribution. Details on this

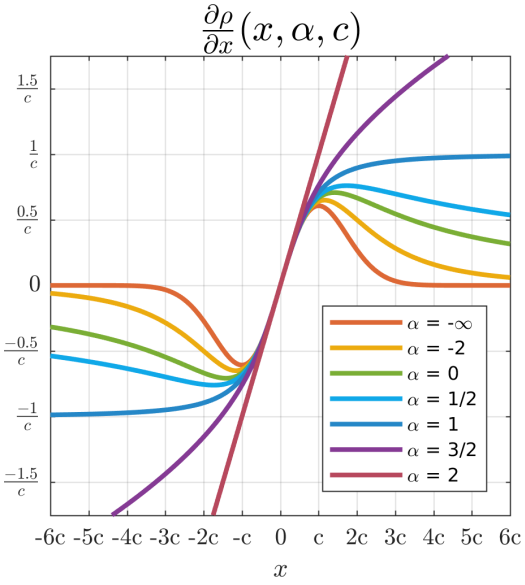

Figure 2.6: The derivative of Equation (2.2) with different α [Barron, 2019].

distribution can be found in origin paper of [Barron, 2019].

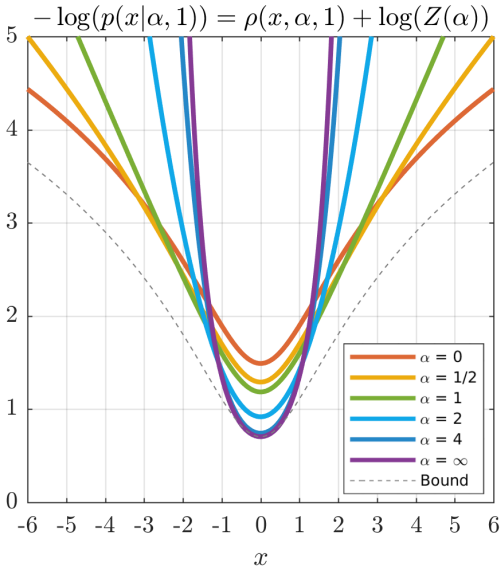

Figure 2.7: Negative log-likelihood distribution of Equation (2.5) [Barron, 2019].

Using the negative log-likelihood of Equation (2.5), if optimisers intend to decrease α to discount outliers, they have to pay extra penalties for inlier data. During the training, optimisers can intelligently determine robustness parameter α. Figure 2.7 plots the negative log-likelihood of Equation (2.5). We see functions that have less penalties for $x \in [-c, c]$ have more penalties for $x \in (-\infty, c) \cup (c, \infty)$. For example, Welsch loss ($\alpha = \infty$) has the lowest penalties for $x \in [-c, c]$, correspondingly has the highest penalties for $x \in (-\infty, c) \cup (c, \infty)$.

2.2.3 Applications

The general and adaptive robust loss function (Equation (2.2)) and its probability density function (Equation (2.5)) achieve state-of-the-art performance on many tasks:

- By replacing the per-pixel normal distribution with per-pixel general distribution (Equation (2.5)) in variational autoencoder models [Kingma and Welling, 2014], [Barron, 2019] improves performance for image synthesis on the CelebA datasets [Liu et al., 2015] .
- [Barron, 2019] replaces the fixed Laplacian distribution with general distribution (Equation (2.5)) in unsupervised monocular depth estimation [Godard et al., 2017]. And the test loss decreases up to 17%.
- Because Geman-McClure loss [Geman and McClure, 1985] is a special instance of Equation (2.2). In fast global registration and robust continuous clustering tasks [Zach, 2014; Zhou et al., 2016b; Shah and Koltu, 2017], [Barron, 2019] replaces Geman-McClure loss with Equation (2.2). The experiments show there is remarkable performance gain.

The decisive advantage of the adaptive and general function (Equation (2.2)) is the ability to generalise different loss functions. During learning, parameters α adjusts to the most appropriate loss functions, and c updates to the best outlier boundary. For different percentage of outliers in input, α and c may converge to different values. Relating to our project, we will formulate the general and adaptive robust loss as a pooling layer in Section 3.2.

2.3 Deep Declarative Networks

Deep learning has achieved astonishing results on large datasets and various tasks. However, many real-world applications require implicit domain knowledge that cannot be modelled as deep learning tasks [Lutter et al., 2019]. Most of these applications can be defined as optimisation problems. There are multiple attempts to integrate optimisation problems into deep learning framework [Gould et al., 2016; Amos and Kolter, 2017; Amos et al., 2018; Chen et al., 2018; Agrawal et al., 2019; Wang et al., 2019]. These state-of-the-arts have shown exceptional effectiveness in solving problems that traditional neural networks fail to handle. For instance, Lagrangian neural networks [Cranmer et al., 2020] and Hamiltonian neural networks [Greydanus et al., 2019] impose physic prior into networks and outperform traditional networks on double pendulum simulations.

Deep declarative networks (DDN) is a generalisation of various previous efforts. Instead of defining the forward function explicitly, the forward process is defined by desired behaviours [Gould et al., 2020]. In DDN, the forward passing involves solving a mathematical optimisation problem. And the backward passing utilises implicit function theorem to derive gradients. An essential benefit of DDN is the capability to co-exist with regular neural networks. This advantage enables researchers

to extend neural networks into geometric model fitting, model-predictive control algorithms, optimal transport, and structured prediction problems [Gould et al., 2020]. DDN can introduce new features to networks as well. Such instances include robust pooling and feature projections, which are the frameworks for our project.

The backward passing of DDN relies on **implicit function theorem** [Steven and Harold, 2013]. Implicit functions are functions defined by implicit equations, which associate different variables. For example, a unit circle (Figure 2.8) can be expressed as:

$$x^2 + y^2 = 1. \tag{2.6}$$

If we are interested to find the explicit relation between x and y, namely

$$y = h(x), \tag{2.7}$$

we cannot find one because for any $x \in (-1, 1)$ there are two corresponding y:

$$\begin{aligned} y &= \sqrt{x^2 - 1} \\ y &= -\sqrt{x^2 - 1}. \end{aligned} \tag{2.8}$$

However, we are able to identify the part of function graph for Equation (2.6): if

$$\begin{aligned} g &= \sqrt{x^2 - 1} \\ x &\in (-1, 1), \end{aligned} \tag{2.9}$$

then we know g represents the upper half of the circle. And similarly, the negation of g represents the lower half of the circle. Implicit function theorem indicates functions like g exist, even there is no explicit formula.

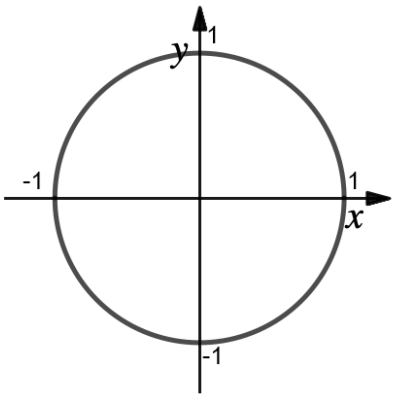

Figure 2.8: Unit circle

Implicit differentiation differentiates through implicitly defined functions. In

Equation (2.6), let

$$f = x^2 + y^2 - 1 \tag{2.10}$$

and assign $f = 0$. We can calculate the deviate of y respect to x when y is not 0:

$$\begin{aligned} &\mathrm{D}_x f + \mathrm{D}_y f = 0 \\ &2x + 2y\mathrm{D}_x y = 0 \\ &\mathrm{D}_x y = -\tfrac{x}{y} \ (y \neq 0). \end{aligned} \tag{2.11}$$

For some implicit functions like Equation (2.6), although we cannot define functions themselves explicitly, we can derive explicit expressions for their gradients.

The forward passing of DDN is a black-box process: we know there exist algorithms to compute the optimal solutions but do not know the mechanism of the algorithms. In the back passing, we use implicit differentiation to retrieve the input gradients with respect to output. At the abstract level, DDN is a bi-level optimisation problem. **Bi-level optimisation** is an optimisation that one problem is embedded into another one. It is divided into high-level and low-level optimisation. The general form of bi-level optimisation is [Gould et al., 2020; von Stackelberg et al., 2011; Bard, 1998]:

$$\begin{aligned} &\text{minimize} \quad J(x, y) \\ &\text{subject to} \quad y \in \arg\min_{u \in C} f(x, u), \end{aligned} \tag{2.12}$$

where the minimisation $J(x, y)$ is the high-level optimisation. It may refer to the sum of loss functions and regularisation. Specifically, high-level optimisation may connect to y via other layers, where y is a possibly implicit function of x. The arg min term is the low-level optimisation that defines behaviours and constraints. To solve problem Equation (2.12), we derive gradients via chain rule and implicit function theorem:

$$\mathrm{D}J(x, y) \quad = \mathrm{D}_X J(x, y) + \mathrm{D}_Y J(x, y)\mathrm{D}y(x), \tag{2.13}$$

where $\mathrm{D}_X J(x, y)$ and $\mathrm{D}_Y J(x, y)$ are partial derivative of J for x and y. The last term contains $\mathrm{D}_Y J(x, y)$ and $\mathrm{D}y(x)$ because y is a function of x [Gould et al., 2020]. The core contribution of DDN theory is the calculation of $\mathrm{D}y(x)$.

2.3.1 Theory of Deep Declarative Networks

[Gould et al., 2020] introduced terms to distinguish declarative networks from conventional networks. The forward passing function in conventional networks is defined explicitly:

$$y = f(x, \theta), \tag{2.14}$$

where θ is the network parameter (possibly empty). In DDN concepts, such nodes are called **imperative nodes**. Function relation $y(x)$ is explicitly defined. Nodes in

DDN are defined by behaviours:

$$y \in \underset{u \in C}{\arg\min}\, f(x, u; \theta), \tag{2.15}$$

where f is the parametric objective function, θ are the parameters, and C are the constraints. The function relation $y(x)$ is implicitly defined. Such nodes are called **declarative nodes**. We will use the term declarative node frequently in the following chapters. Figure 2.9 and Figure 2.10 illustrate the abstractions of two kinds of nodes.

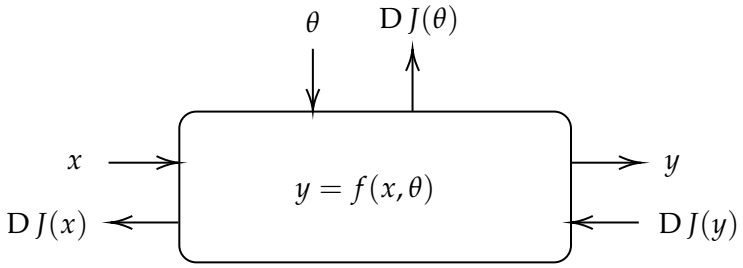

Figure 2.9: Imperative node [Gould et al., 2020]

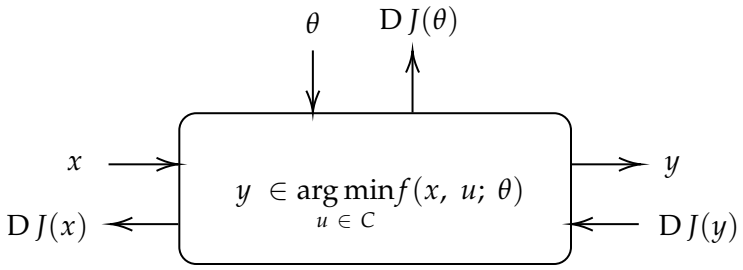

Figure 2.10: Declarative node [Gould et al., 2020]

[Gould et al., 2020] proofs imperative nodes $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$ can be reduced to declarative nodes in the form:

$$y \in \underset{u \in \mathbb{R}^m}{\arg\min}\, \frac{1}{2}\|u - \tilde{f}(x)\|^2. \tag{2.16}$$

This proposition facilitates us to substitute imperative nodes in conventional networks with declarative nodes. We will use Equation (2.16) several times in the following chapters.

2.3.2 Learning in Declarative Nodes

The derivative computation of declarative nodes utilises implicit function theorem from [Dontchev and Rockafellar, 2014]. [Gould et al., 2020] summarised three common sub-classes:

1. The first one is unconstrained case:

$$y \in \underset{u \in \mathbb{R}^m}{\arg\min}\, f(x,u) \tag{2.17}$$

where input u is in m-dimensional space. We assume $y(x)$ exist and function f is second-order differentiable. The gradient for Equation (2.17) is:

$$\mathrm{D}y(x) \quad = -H^{-1}B \tag{2.18}$$

where $H = \mathrm{D}^2_{YY} f(x, y(x)) \in \mathbb{R}^{m\times m}$, $B = \mathrm{D}^2_{XY} f(x, y(x)) \in \mathbb{R}^{m\times n}$ and H is a non-singular matrix. Our adaptive robust pooling in Chapter 3 is an unconstrained declarative node.

2. The second one is equality constrained declarative node:

$$\begin{array}{rll} y(x) \in & \arg\min_{u\in\mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0, \quad i = 1,\ldots,p. \end{array} \tag{2.19}$$

where $h = [h_1, \ldots, h_p]$ and the arg min is constrained by p constraints. We also assume $y(x)$ exist. The gradient is

$$\mathrm{D}y(x) \quad = H^{-1}A^{\mathsf{T}}\big(AH^{-1}A^{\mathsf{T}}\big)^{-1}\big(AH^{-1}B - C\big) - H^{-1}B \tag{2.20}$$

where

$$\begin{array}{ll} A & = \mathrm{D}_Y h(x,y) \in \mathbb{R}^{p\times m} \\ B & = \mathrm{D}^2_{XY} f(x,y) - \sum_{i=1}^p \lambda_i \mathrm{D}^2_{XY} h_i(x,y) \in \mathbb{R}^{m\times n} \\ C & = \mathrm{D}_X h(x,y) \in \mathbb{R}^{p\times n} \\ H & = \mathrm{D}^2_{YY} f(x,y) - \sum_{i=1}^p \lambda_i \mathrm{D}^2_{YY} h_i(x,y) \in \mathbb{R}^{m\times m} \\ \lambda & \in \mathbb{R}^p \text{ satisfies } \lambda^{\mathsf{T}} A = \mathrm{D}_Y f(x,y) \end{array}$$

and H is a non-singular matrix. We assume f, h are second-order differentiable and $\mathrm{D}_y h(x,y)$ is full rank. The adaptive feature projections in Chapter 4 are classified as equality constrained declarative nodes.

3. The last sub-class of declarative node is inequality constrained:

$$\begin{array}{rll} y(x) \in & \arg\min_{u\in\mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0,\ i = 1,\ldots,p \\ & & g_i(x,u) \le 0,\ i = 1,\ldots,q. \end{array} \tag{2.21}$$

Similarly as equality constrained case, $h = [h_1, \ldots, h_p]$ are set of equality constraints and $y(x)$ exists. In addition, $g = [g_1, \ldots, g_p]$ are set of inequality constraints. The gradient is

$$\mathrm{D}y(x) \quad = H^{-1}A^{\mathsf{T}}\big(AH^{-1}A^{\mathsf{T}}\big)^{-1}\big(AH^{-1}B - C\big) - H^{-1}B \tag{2.22}$$

where

$$\begin{aligned} A &= \mathrm{D}_Y \tilde{h}(x,y) \in \mathbb{R}^{(p+q)\times m} \\ B &= \mathrm{D}^2_{XY} f(x,y) - \textstyle\sum_{i=1}^{p+q} \lambda_i \mathrm{D}^2_{XY} \tilde{h}_i(x,y) \in \mathbb{R}^{m\times n} \\ C &= \mathrm{D}_X \tilde{h}(x,y) \in \mathbb{R}^{(p+q)\times n} \\ H &= \mathrm{D}^2_{YY} f(x,y) - \textstyle\sum_{i=1}^{p+q} \lambda_i \mathrm{D}^2_{YY} \tilde{h}_i(x,y) \in \mathbb{R}^{m\times m} \\ \lambda &\in \mathbb{R}^{p+q} \text{ satisfies } \lambda^{\mathsf{T}} A = \mathrm{D}_Y f(x,y) \\ &\qquad \text{with } \lambda_i \leq 0 \text{ for } i = p+1,\ldots,p+q \end{aligned}$$

and f, h and g are second-order differentiable. The matrix $[h_1,\ldots,h_p,g_1,\ldots,g_p]$ is full rank.

[Gould et al., 2020] has included detailed proofs for gradient computations of all three cases and some implementation consideration. Their full Python Pytorch implementation can be found at `http://deepdeclarativenetworks.com`. The code of our project is based on their work.

2.3.3 Simple Example

We borrow an example from [Gould et al., 2016] for demonstration the correctness of DDN theory. We define

$$y(x) \in \quad \arg\min_{u\in\mathbb{R}} f(x,u) \tag{2.23}$$

where

$$f(x,u) = xu^4 + 2x^2u^3 - 12u^2$$

and solve it analytically. The first step is to solve the function derivative at stationary points to get local minimum:

$$\begin{aligned} &Df(x,u)_u = 4xu^3 + 6x^2u^2 - 24u = 0 \\ &u \in \left\{0, \tfrac{-3x^2-\sqrt{9x^4+96x}}{4x}, \tfrac{-3x^2+\sqrt{9x^4+96x}}{4x}\right\}. \end{aligned} \tag{2.24}$$

The gloabal minimum of $f(x,u)$ reaches at one of the three points. After evaluation, it reaches at

$$y = \tfrac{-3x^2-\sqrt{9x^4+96x}}{4x}. \tag{2.25}$$

We can compute Dy(x) by hand as:

$$Dy(x) = -\frac{3x\sqrt{9x^4+96x} + 9x^3 - 48}{4x\sqrt{9x^4+96x}}. \tag{2.26}$$

Then we solve Equation (2.23) with DDN theory. The problem is classified as an unconstrained declarative node (Equation (2.17)). The gradient computation uses

Equation (2.18):

$$\begin{aligned} H &= (12xy^2 + 12x^2y - 24) \\ B &= (4y^3 + 12xy^2) \\ Dy(x) &= -\left(12xy^2 + 12x^2y - 24\right)^{-1} \left(4y^3 + 12xy^2\right) \\ &= -\frac{3x\sqrt{9x^4 + 96x} + 9x^3 - 48}{4x\sqrt{9x^4 + 96x}}. \end{aligned} \tag{2.27}$$

We obtained the same expression for $Dy(x)$ in Equation (2.26) and Equation (2.27) using explicit and implicit (DDN) approaches respectively. Hence we verified the correctness of DDN theory in a simple example.

2.3.4 Future Work

DDN is a relatively new research area, and therefore there are still a lot to explore. From the theoretical perspective, DDN with parametric constraints may run into infeasibility in backpropagation [Gould et al., 2020]. In our project, we will discuss how to tackle this issue in particular instances. Moreover, DDN with multiple constraints and non-regular solutions does not have comprehensive theories at the current stage ([Wang, 2020] provides some insightful study on these topics).

From the application perspective, the initial applications of DDN are robust pooling and feature projections, which built the frameworks of this project and will be discussed in detail later. Before and concurrent to my project, other researchers extend DDN to multiple disciplines. These topics include adversarial attack, rank pooling, robust batch normalisation and conditional random fields. In future, we believe DDN can solve more challenges facing by the machine learning community.

Chapter 3

Adaptive Robust Pooling

In this chapter, we first review robust pooling from [Gould et al., 2020] and motivate its adaptive version in Section 3.1. Besides, we formulate the general and adaptive robust loss function from [Barron, 2019] as a pooling layer in Section 3.2. In Section 3.3, we then derive the derivative formulas for our methods. Finally, we conduct experiments on our methods in Section 3.4 and 3.5.

Robust pooling operation was introduced as an instance of DDN in [Gould et al., 2020]. For completeness, we briefly walk through the deviation again. Average pooling, usually used as a downsampling method in CNN, is defined as:

$$y = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{3.1}$$

In declarative form, Equation (3.1) can be rewritten as an arg min of quadratic loss (also called squared loss):

$$y \in \arg\min_{u \in \mathbb{R}} \sum_{i=1}^{n} \frac{1}{2}(u - x_i)^2. \tag{3.2}$$

Average pooling is sensitive to outliers. To improve robustness, we generalise Equation (3.2) as a function ϕ with additional parameter c

$$y \in \arg\min_{u \in \mathbb{R}} \sum_{i=1}^{n} \phi(u - x_i, c) \tag{3.3}$$

where c controls the threshold between inliers and outliers. With the additional parameter c, we can define $\phi(z, c)$ in Equation (3.3) explicitly with several robust loss functions:

1. **Huber loss** [Huber, 1964] is defined pairwise: it is quadratic for values that are less or equal to c, and linear for values that are greater than c. The function is motivated by squared loss and absolute loss:

$$\phi(z, c) = \begin{cases} \frac{1}{2}z^2 & \text{for } |z| \leq c \\ \alpha(|z| - \frac{1}{2}c) & \text{otherwise.} \end{cases} \tag{3.4}$$

2. **Pseudo-Huber loss** [Charbonnier et al., 1997] is a smoother approximation of

Huber loss. It is less steep for extreme value than Huber loss. The derivative of pseudo-Huber loss is continuous in all degree:

$$\phi(z,c) = c^2\left(\sqrt{1+\left(\frac{z}{c}\right)^2}-1\right). \tag{3.5}$$

3. **Welsch loss** [Dennis and Welsch, 1978] was developed when solving nonlinear least squares and robust regression problems:

$$\phi(z,c) = 1-\exp\left(-\frac{z^2}{2c^2}\right). \tag{3.6}$$

4. **Truncated quadratic loss** [Gould et al., 2020] is a variation of quadratic loss function. For inliers, the loss function behaviours identical to the quadratic loss function. For outliers, the cost is a fixed constant:

$$\phi(z,c) = \begin{cases} \frac{1}{2}z^2 & \text{for } |z| \le c \\ \frac{1}{2}c^2 & \text{otherwise.} \end{cases} \tag{3.7}$$

Figure 3.1 (left) plots the graph of these robust functions along with quadratic loss function. We can observe all functions overlap for the inlier range $[-c, c]$. Only quadratic loss applies the same penalty scheme for outliers and inliers. Figure 3.1 (left) is very similar to Figure 2.5 in Section 2.2, which verifies that the adaptive and general robust loss function (Equation (2.2)) can resemble several loss functions. Figure 3.1 (right) shows the sum of the two functions centered at $x = 2c$ and $x = -2c$. We observe quadratic-Huber, Huber and quadratic loss are convex functions; Welsch and truncated quadratic loss are not convex.

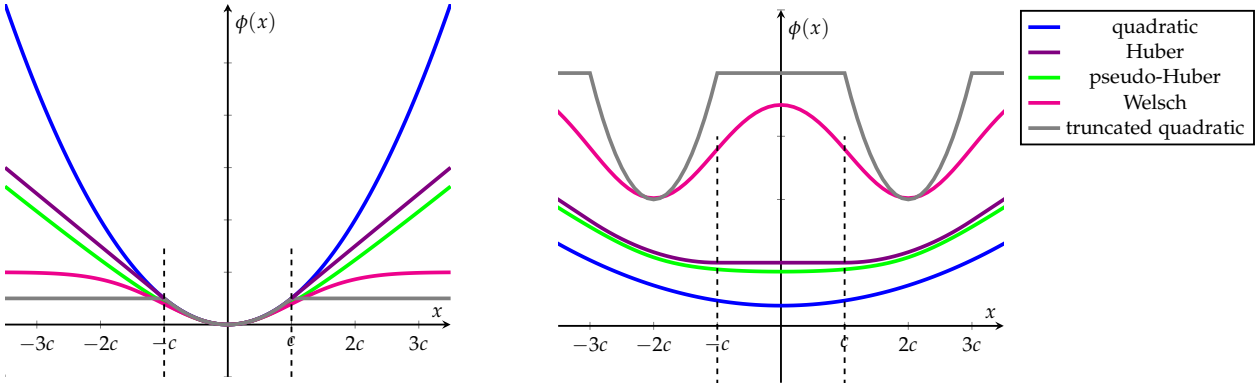

Figure 3.1: **Left:** quadratic, pseudo-Huber, Huber, Welsch and truncated quadratic loss. c defines outlier threshold. **Right:** sum of two penalty functions centered at $x = 2c$ and $x = -2c$ [Gould et al., 2020]. The y axis has been scale for visualisation.

We will continue to use Huber, pseudo-Huber and Welsch loss in succeeding sections. Quadratic and truncated quadratic loss do not apply in this project's scope, and we will explain the reason in Section 3.3. The arg min expression in Equation (3.3) is

called **robust pooling** and is classified as an unconstrained case (Equation (2.17)) of declarative node. [Gould et al., 2020] experimented with robust pooling on PointNet: when the input point clouds carry outliers, robust pooling produces more favourable results than max pooling. Their experiments are our baselines in Section 3.4.

3.1 Definition

Parameter c in Equation (3.3) decides whether each feature is inlier or outlier. In the PointNet experiments, [Gould et al., 2020] set threshold c as constant 1 (and already obtained extraordinary performance gain). One may ask what if $c = 2$, $c = 3$ or $c = 100$? How the value of c seizes model behaviours? What is the most desirable value of c for different models and datasets?

We can conduct extensive parameter searching and tuning, then set c as a fixed constant. Nevertheless, the procedure is computationally expensive and hard to migrate to large-scale tasks. In this project, we instead determine **a suitable value of c from networks**: give c a primary value and update this value in end-to-end backpropagation. Relating to the background in Section 2.1.2, we are shifting robust pooling from non-parametric network layers to parametric network layers.

If $c \leq 0$, all input in Equation (3.3) is classified as outliers. Hence we need to enforce a constraint on c:

$$\begin{aligned} &y \in argmin \sum_{i=1}^{n} \phi(u - x_i, c) \\ &\text{subject to } c > 0. \end{aligned} \tag{3.8}$$

The constraint in Equation (3.8) transfers the problem from an unconstrained declarative node (Equation (2.17)) to an inequality constrained declarative node (Equation (2.19)). The transformation brings us to two challenges:

- We have to apply more complicated rules (Equation (2.20)) for the gradient calculation.
- Because Equation (3.8) contains a parametric constraint, the problem may become infeasible during learning [Gould et al., 2020].

We avoid these two difficulties by applying an affine softplus transformation (Figure 3.2) to c. The affine softplus is defined as

$$f(x) = \quad (l - m)(1 + exp(x + g^{-1}(s))) + m \tag{3.9}$$

where $g(x) = (1 + exp(x))$ and g^{-1} is the inverse of function of $g(x)$. m, l and s are constants. We set $m = 1$. we set $l = 0$ as the lower bound for $f(x)$. we set $s = 1$ as the shifting value of softplus. This method is inspired by [Barron, 2019].

In networks, we apply $f(x)$ to c every time it feeds into the robust pooling layers. Accordingly, we guarantee the robust pooling problem is feasible since transferred c is always positive. And we avoid using a more complicated gradient calculation rule

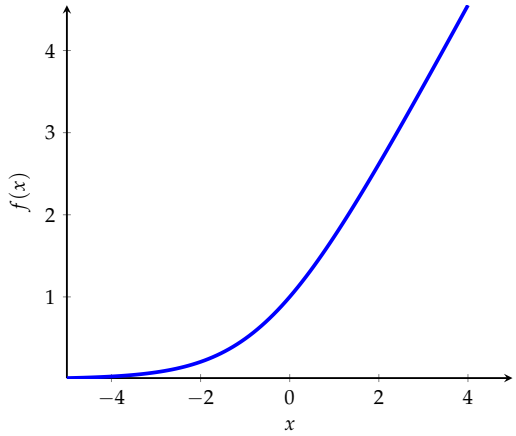

Figure 3.2: Affine softplus transformation

by removing the constraint on c. Our **adaptive robust pooling** is a composition of two layers:

Input:	x
Parameter:	c^*
Transformation Layer:	$c = f(c^*)$
	where f is affine softplus defined in Equation (3.9)
Declarative Layer:	$y \in argmin \sum_{i=1}^{n} \phi(u - x_i, c)$
	where ϕ is pseudo-Huber (Equation (3.4)), Huber (Equation (3.5))
	or Welsch (Equation (3.6))
Forward Output:	y
Backward Output:	$D_X y(x, c^*), D_{C^*} y(x, c^*)$

(3.10)

[Gould et al., 2020] derived the gradient $D_X y(x, c^*)$. In Section 3.3.1, we demonstrate the calculation of $D_{c^*} y(x, c^*)$.

The workflow of adaptive robust pooling is shown in Figure 3.3. Theoretically, we attempt to learn c as the best boundary that the robust mean is close to the actual mean (without outliers). However, because features are transformed multiple times before feeding into pooling layers, this effect is hard to verify in deep networks. Therefore, before embedding adaptive robust pooling into deep networks and testing it on a large dataset, we want to validate it on a toy example.

3.1.1 Bi-level Optimisation Example

We demonstrate adaptive robust pooling on a bi-level optimisation with one-dimensional data. Instead of estimating the average of features as in neural networks, we measure the average from outlier-contained data. We use Huber pooling as an example.

Initialised with a random c, our objective is to determine appropriate c that defines the boundary between outliers and inliers. Figure 3.4 (left) shows some ran-

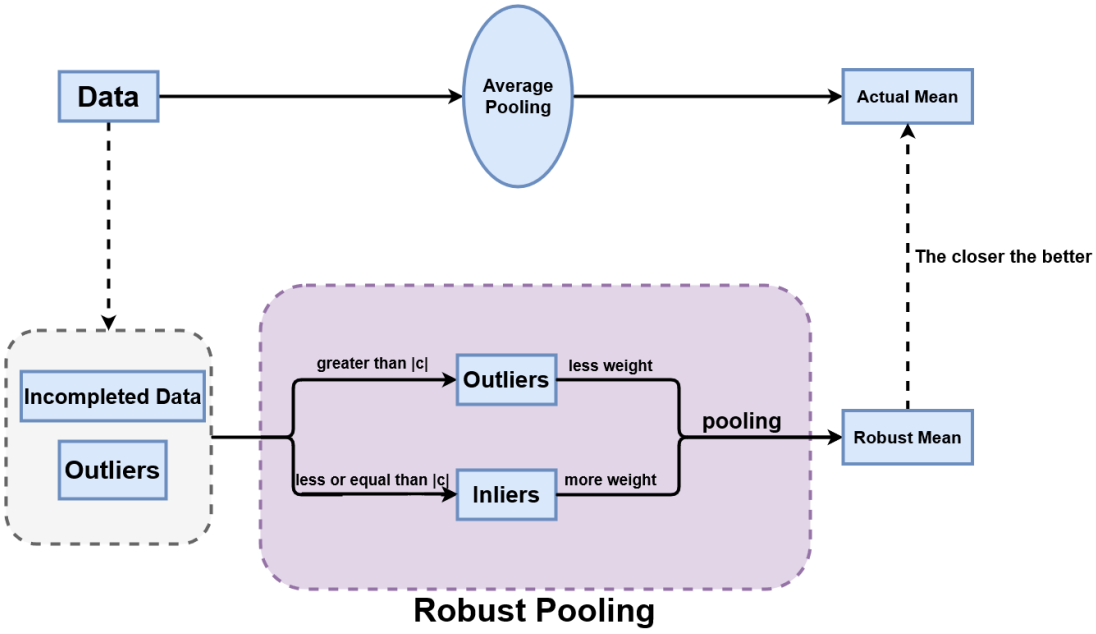

Figure 3.3: Work flow of adaptive robust pooling

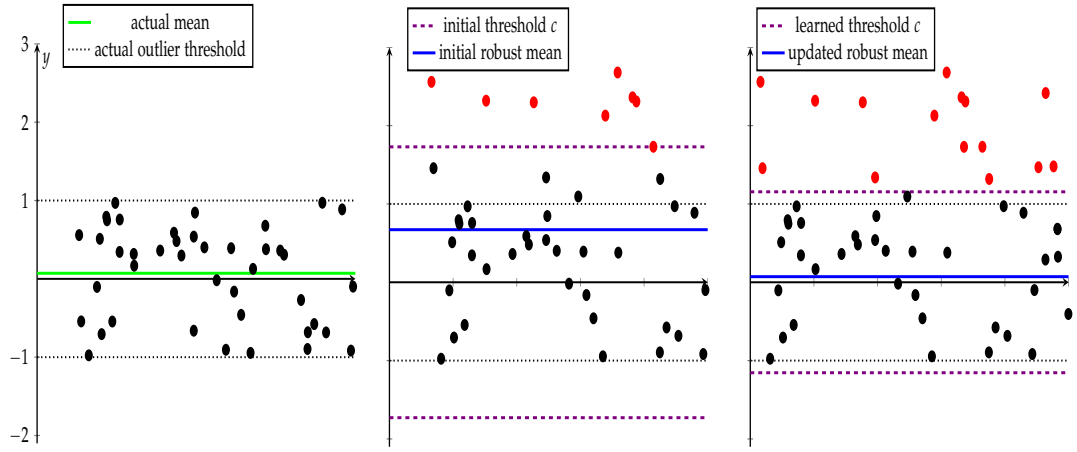

Figure 3.4: Toy example in bi-level optimisation. **Left**: 50 points in range $[-1, 1]$ are sampled in the y-axis direction and spread across the x-axis direction for visualisation. The mean of samples is shown in the green line. **Mid**: 20 points are replaced with outliers in range $[1, 3]$. An initial outlier threshold c in sampled $|c| \in [1, 3]$ and shown in violet dashed line. Red points are classified as outliers. The estimated robust mean with Huber pooling is shown blue line. **Right**: using the mean in the leftmost figure as ground truth and estimated mean in mid figure as output, we learn c using mean squared loss. The violet dashed line is the updated c, and the blue line is the robust mean estimated from updated c.

domly sampled points and their average. In the mid and right of Figure 3.4, some outliers were added along the y axis. In Figure 3.4 (mid), because the randomly

picked threshold c is far from the actual boundary, the pooling employs heavy penalties to some outliers (points lie between the top violet and black lines). Consequently, we notice that the estimated mean is far from the actual mean in the leftmost figure.

We update the value of c with the actual mean: in the machine learning concept, the estimated robust mean in Figure 3.4 (mid) is considered output prediction, and the actual mean in Figure 3.4 (left) is recognised as the ground truth. We calculate the error between them and backpropagate. c is updated with the gradient descent method. Analysing the mid and right of Figure 3.4, we see the updated c is near the actual boundary, and the predicted robust mean closes to the actual mean. Accordingly, we confirmed that adaptive robust pooling could decide a reasonable boundary in a simple model.

3.2 General and Adaptive Robust Loss as Pooling

Pseudo-Huber, Huber and Welsch loss functions contain one parameter c controlling the boundary between inliers and outliers. The penalty schemes for outliers and inliers are pre-determined by the function definitions, usually associated with convexity. On the contrary, the innovation function reviewed in Section 2.2 has the flexibility to decide the penalty scheme by adjusting parameter α. The function definition is given by:

$$f(x, \alpha, c) = \frac{|\alpha-2|}{\alpha}\left(\left(\frac{(\frac{x}{c})^2}{|\alpha-2|}+1\right)^{\frac{\alpha}{2}}-1\right). \tag{3.11}$$

3.2.1 Linear Regression Example

Before proceeding to our approach, we delve deeper into Equation (3.11) for a better understanding. We fit the loss function with a linear regression model:

$$y = wx + b. \tag{3.12}$$

We first uniformly sample 50 1-dimensional points over interval $[0, 1]$ as x. We then generate true y with some randomly picked w_0 and b_0 using Equation (3.12) (take x as input). We write true y as y^*. Moreover, we randomly sample 10 elements from y^* and replaced them with outliers. Our goal is to learn w_0 and b_0 using learn regression model: x is input, w_1 and b_1 are randomly initlised model weight, and y^* is the ground truth.

We train our linear regression model with Equation (3.11) and mean squared loss. The latter one is a frequently-used non-robust loss function. In adaptive robust loss, we have three different settings: α is learnable parameter and c is constant; c is learnable parameter and α is constant; both α and c are learnable parameters. We plot the learned linear function in Figure 3.5.

In Figure 3.5, we observe that function learned with the mean squared loss deviates a lot, indicating that the mean squared loss is susceptible to outliers. The learned line for learnable α in Equation (3.11) is essentially identical to the mean squared loss

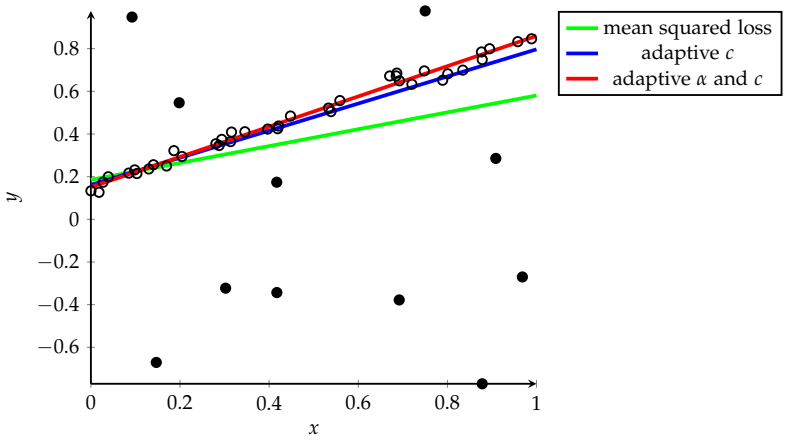

Figure 3.5: Outliers are presented by filled dots and inliers are unfilled circle. The green line is learned via mean squared loss, blue line is learned via Equation (3.11) with learnable c, the red line is learned by setting both α and c learnable in Equation (3.11). The ground truth is overlapping with red line hence we did not plot it.

one and hence we did not plot it. It shows making α learnable does not improve robustness. Furthermore, we see the learned function for adaptive c in Equation (3.11) is closer to the actual function line, proving that parametric c increases model robustness. The learned function with learnable α and c in Equation (3.11) overlaps with ground truth and yields the best result. To summarise, the general and adptive loss (Equation (3.11)) is more robust to outliers in a simple regression model.

3.2.2 Definition

We have gained a sense of 'robustness' in the previous example. Now we continue to present our method. We can express Equation (3.11) in arg min as pooling:

$$y \in \arg\min_{u \in \mathbb{R}} \sum_{i=1}^{n} \frac{|\alpha - 2|}{\alpha} \left(\left(\frac{\left(\frac{u - x_i}{c}\right)^2}{|\alpha - 2|} + 1 \right)^{\frac{\alpha}{2}} - 1 \right) \tag{3.13}$$

Comparing Equation (3.13) with pseudo-Huber (Equation (3.4)), Huber (Equation (3.5)) and Welsch (Equation (3.6)), we see there is an additional α for robustness control. Our hope is that the parameter α delivers more flexibility and interpretability: it tunes a proper outlier threshold c and employs an adjustable penalty scheme α simultaneously.

Although in theory α can be any number in $\mathbb{R}$, [Barron, 2019] states α should be in range $[0, 3]$ to avoid numerical instability in applications. In this range, Equation (3.13) is a generalisation of pseudo-Huber ($\alpha = 1$) and quadratic ($\alpha = 0$) pooling.

If we parameterise α and c, we obtained an inequality constrained declarative node:

$$\begin{array}{rll} y \in & \arg\min_{u \in \mathbb{R}} & \sum_{i=1}^{n} \phi(u - x_i, c, \alpha) \\ & \text{subject to} & c > 0 \\ & & 0 \leq \alpha \leq 3 \end{array} \tag{3.14}$$

where arg min equation is defined in Equation (3.13). We encountered the similar

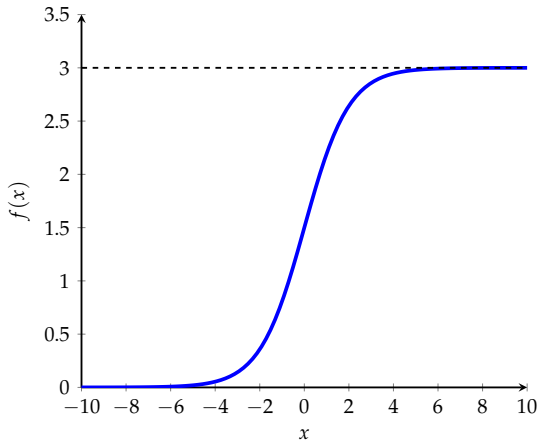

Figure 3.6: Affine sigmoid transformation

problem in adaptive robust pooling. To eliminate constraints in Equation (3.14), we employ affine softplus (Equation (3.9)) to c as in Section 3.1. We then apply affine sigmoid function (Figure 3.6) to enforce the constraint on α:

$$f_2(x) = \quad g_2(x)(M - m) + m \tag{3.15}$$

where

$$g_2(x) = \tfrac{1}{1+e^{-x}}.$$

We set $m = 0$ as the lower bound and $M = 3$ as the upper bound. Accordingly, our two-layer **general and adaptive robust pooling** is defined as:

$$
\begin{array}{ll}
\text{Input:} & x \\
\text{Parameters:} & \alpha^*, c^* \\
\text{Transformation Layer:} & c = f_1(c^*) \\
 & \quad \text{where } f_1 \text{ is affine softplus defined in Equation (3.9)} \\
 & \alpha = f_2(\alpha^*) \\
 & \quad \text{where } f_2 \text{ is affine sigmoid defined in Equation (3.15)} \\
\text{Declarative Layer:} & y \in \arg\min_{u \in \mathbb{R}} \sum_{i=1}^{n} \phi(u - x_i, c, \alpha) \\
 & \quad \text{where } \phi(u - x_i, c, \alpha) = \frac{|\alpha-2|}{\alpha}\left(\left(\frac{(\frac{u-x_i}{c})^2}{|\alpha-2|} + 1\right)^{\frac{\alpha}{2}} - 1\right) \\
\text{Forward Output:} & y \\
\text{Backward Output:} & \mathrm{D}_X y(x, \alpha^*, c^*), \mathrm{D}_\alpha y(x, \alpha^*, c^*), \mathrm{D}_C y(x, \alpha^*, c^*)
\end{array}
\tag{3.16}
$$

The exact expressions for $\mathrm{D}_X y(x, \alpha^*, c^*), \mathrm{D}_\alpha y(x, \alpha^*, c^*)$ and $\mathrm{D}_C y(x, \alpha^*, c^*)$ will be discussed in next section.

3.3 Gradients

In prior sections, we developed (general and) adaptive robust pooling as two-layer declarative nodes. We enforced feasibility and avoided complex gradient computation by using affine transformations. In our two-layer pooling, we can handle the gradients of the transformation layer with the Pytorch AutoGrad library [Paszke et al., 2017]. Hence there is no need to derive the exact derivative formulas. The gradients for the declarative nodes requires applying DDN theory. In our cases, we have unconstrained declarative nodes (Equation (2.17)), and we calculate gradients using (Equation (2.18)). The complete steps following styles in [Gould et al., 2020] are given in Appendix A.1.

3.3.1 Adaptive Robust Pooling

We first compute gradients of adaptive pseudo-Huber, Huber and Welsch pooling (Equation (3.4), Equation (3.5) and Equation (3.6)). Quadratic pooling is not discussed here because it does not contain parameter c; truncated quadratic pooling (Equation (3.7)) is not discussed here because $\mathrm{D}_C y(x, c) = 0$. The gradient $\mathrm{D}_X y(x, c)$ is identical to the work in [Gould et al., 2020]. Our work focus on $\mathrm{D}_C y(x, c)$:

- Pseudo-Huber:

  arg min objective:

$$f(x,c,y) = \sum_{i=1}^{n} c^2 \left(\sqrt{1 + \left(\frac{y-x_i}{c}\right)^2} - 1 \right)$$

  Gradients:

$$\begin{aligned} \mathrm{D}_X y(x,c) &= \mathbf{vec}\left\{ \frac{w_i}{\sum_{j=1}^{n} w_j} \right\}^T \\ &\quad \text{where } w_i = \left(1 + \left(\frac{y-x_i}{c}\right)^2\right)^{-3/2} \\ \mathrm{D}_C y(x,c) &= \sum_{i=1}^{n} \sum_{j=1}^{n} ((1 + \tfrac{y-x_i}{c})^2)^{\frac{3}{2}} \left(\frac{(y-x_j)^3}{((\frac{y-x_j}{c})^2+1)^{\frac{3}{2}} c^3}\right) \end{aligned} \tag{3.17}$$

- Huber:

  arg min objective:

$$f(x,c,y) = \sum_{i=1}^{n} \begin{cases} \frac{1}{2}(y-x_i)^2 & \text{for } ||y-x_i|| \le c \\ c(|y-x_i| - \frac{1}{2}c) & \text{otherwise} \end{cases}$$

  Gradients:

$$\begin{aligned} \mathrm{D}_X y(x,c) &= \mathbf{vec}\left\{ \frac{[\![|y-x_i| \le c]\!]}{\sum_{j=1}^{n} [\![|y-x_j| \le c]\!]} \right\}^T \\ \mathrm{D}_C y(x,c) &= -\frac{v}{\sum_{i=1}^{n} [\![|y-x_i| \le c]\!]} \\ &\quad \text{where } v = \sum_{i=1}^{n} \begin{cases} 0 & |y-x_i| \le c \\ 1 & y - x_i > c \\ -1 & y - x_i < c \end{cases} \end{aligned} \tag{3.18}$$

- Welsch:

  arg min objective:

$$f(x,c,y) = \sum_{i=1}^{n} (1 - \exp(-\tfrac{(y-x_i)^2}{2c^2}))$$

  Gradients:

$$\begin{aligned} \mathrm{D}_X y(x,c) &= \mathbf{vec}\left\{ \frac{w_i}{\sum_{j=1}^{n} w_j} \right\}^T \\ \mathrm{D}_C y(x,c) &= \frac{v}{\sum_{i=1}^{n} w_i} \\ &\quad \text{where } w_i = \frac{c^2 - (y-x_i)^2}{c^4} \exp\left(-\frac{(y-x_i)^2}{2c^2}\right) \\ &\quad v = \sum_{i=1}^{n} \frac{\left(2(y-x_i)c^2 - (y-x_i)^3\right) exp(-\frac{(y-x_i)^2}{2c^2})}{c^5} \end{aligned} \tag{3.19}$$

Above we derived the gradients for declarative layers. We backpropagate through the transformation layer via chain rule:

$$\mathrm{D}_{C^*} y(x, c^*) = \mathrm{D}_X y(x, c) \cdot \mathrm{D}_{C^*} f_1(c^*) \tag{3.20}$$

where f_1 is the affine softplus function in Equation (3.9). $\mathrm{D}_{c^*} f_1(c^*)$ is acquired via Pytorch AutoGrad [Paszke et al., 2017]. We have obtained the required gradients $\mathrm{D}_X y(x, c^*)$ and $\mathrm{D}_{C^*} y(x, c^*)$ in Equation (3.10).

3.3.2 General and Adaptive Robust Pooling

We then compute gradients for general and adaptive robust pooling in the same style. The difference in this case is we additional need gradient for α. We compute $\mathrm{D}_X y(x, \alpha^*, c^*), \mathrm{D}_\alpha y(x, \alpha^*, c^*)$ and $\mathrm{D}_C y(x, \alpha^*, c^*)$.

arg min objective:

$$\phi(x, \alpha, c, y) = \sum_{i=1}^{n} \left(\frac{|\alpha-2|}{\alpha} \left(\left(\frac{\left(\frac{y-x_i}{c}\right)^2}{|\alpha-2|} + 1 \right)^{\frac{\alpha}{2}} - 1 \right) \right)$$

Gradients:

$$\begin{aligned}
\mathrm{D}_X y(x, \alpha, c) &= -\frac{\mathbf{vec}\{w_i\}}{\sum_{i=1}^{n} w_i} \\
\mathrm{D}_C y(x, \alpha, c) &= -\frac{v}{\sum_{i=1}^{n} w_i} \\
\mathrm{D}_\alpha y(x, \alpha, c) &= -\frac{m}{\sum_{i=1}^{n} w_i} \\
\text{where } w_i &= \left(\frac{|\alpha-2| \left((\alpha-1) z_i^2 + |\alpha-2| c^2 \right) \left(\frac{z_i^2}{|\alpha-2| c^2} + 1 \right)^{\frac{\alpha}{2}}}{\left(z_i^2 + |\alpha-2| c^2 \right)^2} \right) \\
v &= \sum_{i=1}^{n} \sum_{j=1}^{n} \left(\left(\frac{z_i}{\alpha} \right)^2 + 1 \right)^{\frac{3}{2}} \left(\frac{(z_j)^3}{\left(\left(\frac{z_j}{\alpha} \right)^2 + 1 \right)^{\frac{3}{2}} \alpha^3} \right) \\
m &= \sum_{i=1}^{n} \frac{z_i \left(\frac{z_i^2}{c^2|\alpha-2|} + 1 \right)^{\frac{\alpha}{2}-1} \left(\frac{\left(\frac{z_i^2}{c^2|\alpha-2|} + 1 \right)}{2} - t_i \right)}{c^2} \\
t_i &= \frac{z_i^2 \left(\frac{\alpha}{2} - 1 \right)}{c^2 \left(\frac{z_i^2}{c^2|\alpha-2|} + 1 \right) |\alpha-2|(\alpha-2)} \\
z_i &= y - x_i
\end{aligned} \tag{3.21}$$

Using chain rule, we get:

$$\begin{aligned}
\mathrm{D}_{C^*} y(x, \alpha^*, c^*) &= \mathrm{D}_X y(x, \alpha, c) \cdot \mathrm{D}_{C^*} f_1(c^*) \\
\mathrm{D}_{\alpha^*} y(x, \alpha^*, c^*) &= \mathrm{D}_X y(x, \alpha, c) \cdot \mathrm{D}_{\alpha^*} f_2(\alpha^*)
\end{aligned} \tag{3.22}$$

where $\mathrm{D}_{C^*} f_1(c^*)$ and $\mathrm{D}_{\alpha^*} f_2(\alpha^*)$ are gradients of affine softplus and sigmoid. And they are calculated using Pytorch AutoGrad [Paszke et al., 2017]. We can see the gradient formulas is more complex than those in Section 3.3.1. The implementation

of all robust pooling layers includes the forward passing and back passing. Our vectorised Python Pytorch implementation can be found at https://github.com/WenboDu1228/ddn_pooling_and_projections. The implementation was checked with PyTorch AutoGrad and some samples.

3.4 Point Cloud Classification

In this section and Section 3.5, we conduct experiments of previously developed methods on point cloud classification and image classification tasks.

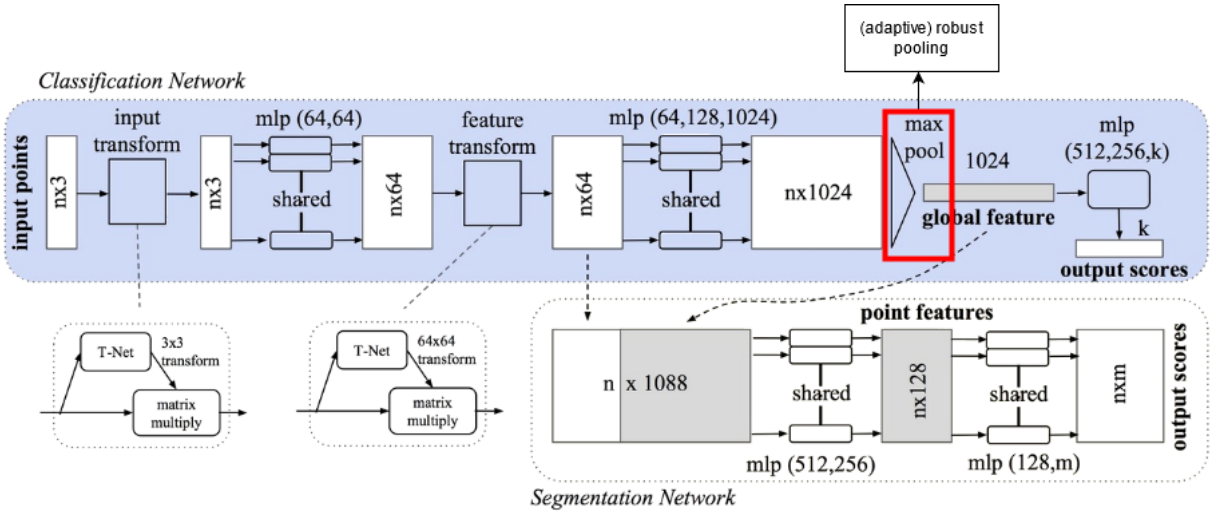

Figure 3.7: PointNet architecture [Qi et al., 2016]

A point cloud is a set point in Cartesian coordinates (X, Y, Z) depicting an object or 3D shape. It is normally created by 3D scanners or photogrammetry software. ModelNet [Wu et al., 2015] is a commonly used dataset for point cloud classification. ModelNet contains some common object categories, and each object is presented as a 2048-point point cloud. ModelNet40 has 40 classes, and ModelNet10 has 10 classes (relatively small). Figure 3.8 (left) and Figure 3.8 (mid) are an example of object in ModelNet10 and its point cloud representation.

Given a set of point clouds and their pre-defined classes, point cloud classification predicts which class each point cloud belongs to based on their geometry attributes. PointNet is a state-of-the-art network architecture for point classification. In particular, it has the advantages of invariance to points permutation [Qi et al., 2016]. We presented the architecture of PointNet in Figure 3.7.

We choose ModelNet40, ModelNet10 as dataset and PointNet as network model. For data preprocessing, we normalise point clouds into unit balls (Figure 3.8 (right)) and replace certain fractions points with outliers. The outliers are sampled from the unit ball as displayed in Figure 3.9. Furthermore, the works in [Gould et al., 2020] are considered as **baselines** of our project: they substituted the global max pooling in PointNet with global robust pooling. This modification has shown remarkable

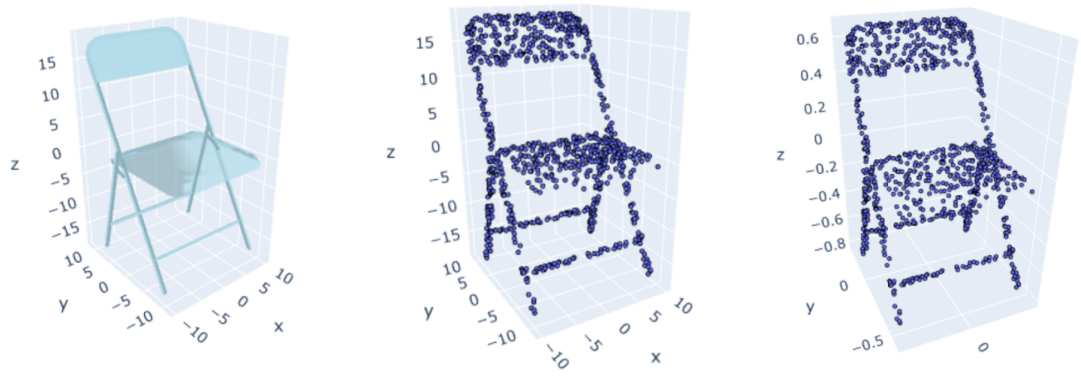

Figure 3.8: **Left**: a chair object in ModelNet10. **Mid:** the point cloud representation of the chair. **Right:** the normalised point cloud representation of the chair.

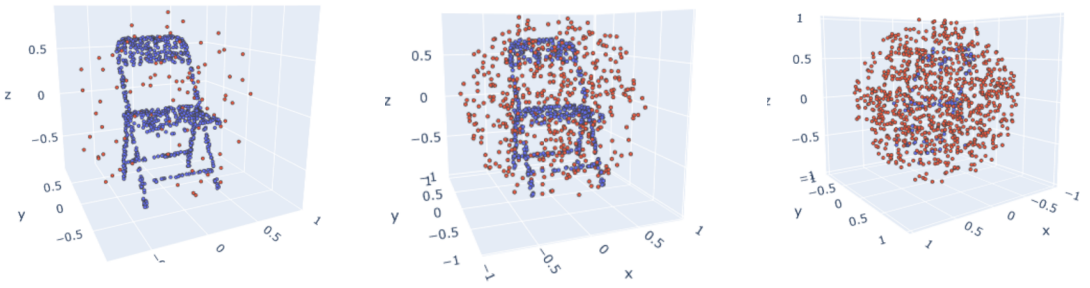

Figure 3.9: **Left to Right**: the chair in point cloud representation with 10%, 50% and 90% of outliers respectfully, the outliers are sampled from a unit ball. The outlier are highlighted in red.

performance gain in point cloud classification. We highlighted their modification in Figure 3.7.

We replace robust pooling layers in baselines with our adaptive versions of pseudo-Huber, Huber and Welsch pooling. The baselines have constant $c = 1$ and in our methods c is initlised as 1. We measure top-1 accuracy and mean average precision. In all experiments, the random seed for outlier generation is the same as in [Gould et al., 2020], hence each run has the same point clouds input. To check reproducibility, we experimented with different seeds multiple times. The results and conclusions are consistent with the reported ones below.

3.4.1 ModelNet40

First, we conduct two experiments on ModelNet40: Figure 3.10 shows results when the outliers are presented in training and testing. In Figure 3.11, the outliers are presented in testing only.

We summarise our discovery as following: whether there are outliers presented

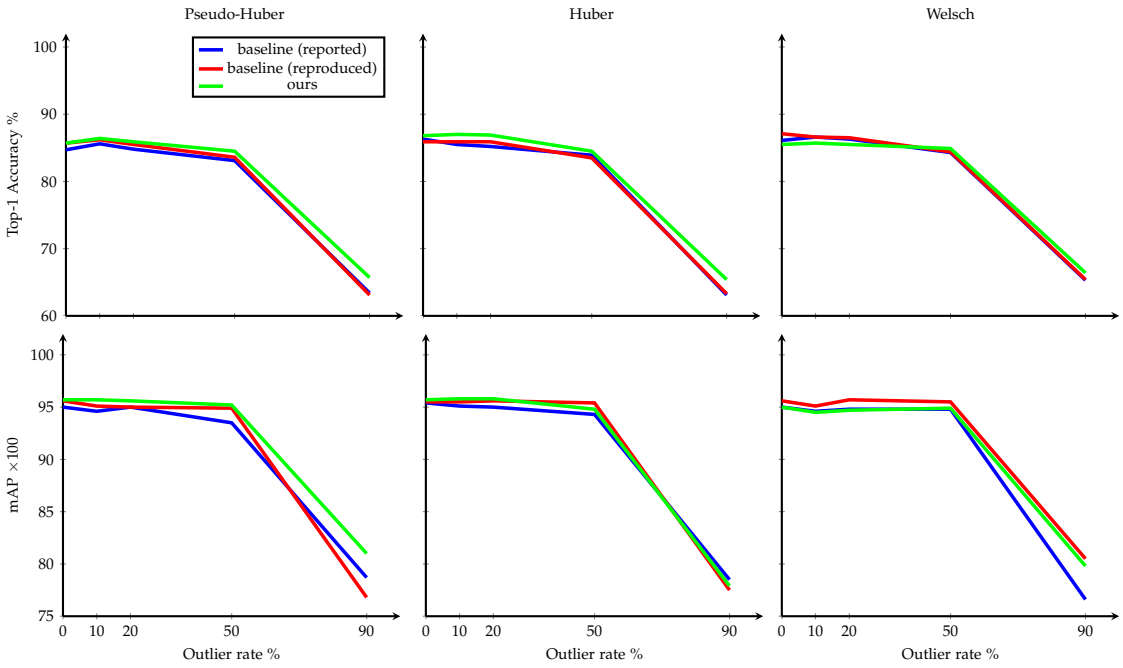

Figure 3.10: The top three figures record Top-1 accuracy, and the buttom three figures records the mean average precision. Outliers are presented in both training and testing. We test adaptive pseudo-Huber, Huber and Welsch pooling. We reported and reproduced results from [Gould et al., 2020] and compare them with our adaptive robust pooling. The y axis does not start from 0. Outlier rates of 0%, 10%, 20%, 50% and 90% are tested.

in training or not, the adaptive robust pooling increases top-1 accuracy and mean average precision (mAP) in most cases. From Figure 3.10 , we see the improvement is very consistent for Huber and pseudo-Huber cases. For Welsch pooling, there is a performance decrease. In Figure 3.11, we see our adaptive pooling start to outperform baselines if there are more than 20% of outliers. The effect is most significant for 50% to 80% of outliers. We additionally plot the results of PointNet with original max pooling to show the significance of robust pooling methods in Figure 3.11 (left).

Figure 3.12 compares the top-1 accuracy of pseudo-Huber pooling with 10% and 90% outliers (outliers presents in training and testing) in every epoch. We can see the accuracy advantages of our methods shows at early epochs. When there are 90% outliers, adaptive methods perform notably better. Similar conclusions can be drawn from other adaptive pooling. However, we do not report them here for simplicity.

During experiments, we found large learning rate causes exploding gradients. This issue is eliminated by decreasing the learning rate from 0.01 to 0.001. Surprisingly, by examining reported and reproduced baselines results in Figure 3.10 and Figure 3.11, this modification improves results in non-adaptive robust pooling too. The exact results for Figure 3.10 and Figure 3.11 are recorded in Appendix B.1.

Max, quadratic, pseudo-Huber, Huber, Welsch and truncated quadratic pooling

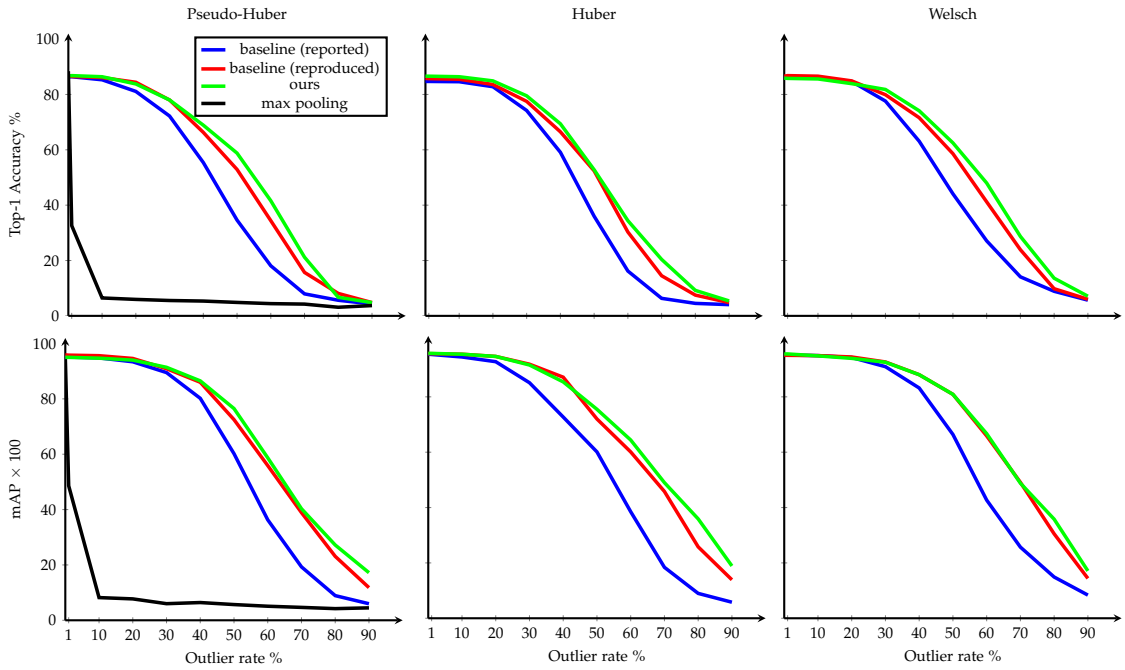

Figure 3.11: Top 3 figures record the top-1 accuracy and buttom figures record mean average precision. Outliers are presented in testing only. We test adaptive pseudo-Huber, Huber and Welsch pooling; we reported and reproduced results from [Gould et al., 2020] and compare them with our methods. 0%, 1%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% and 90% of outliers experimented. We additionally plot the result of max pooling on two leftmost figures.

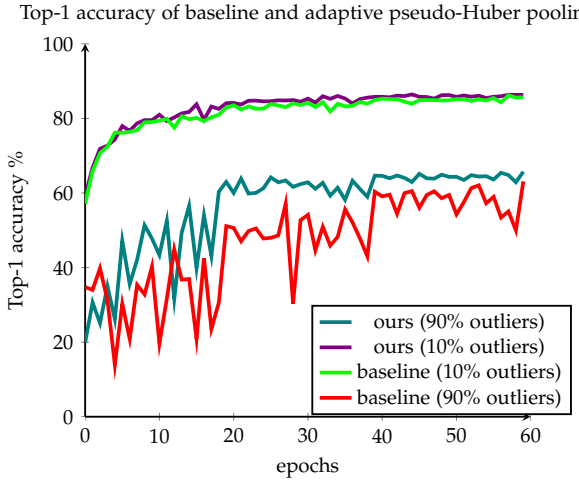

Figure 3.12: Top-1 accuracy over 60 epochs for baselines and our adaptive pseudo-Huber pooling. 10% and 90% of outliers rate are presented in both training and testing.

are evaluated in [Gould et al., 2020]. The results suggested truncated quadratic is in favour in most circumstances. Max, quadratic, and truncated quadratic pooling do not contain learnable parameters, and therefore they are not included in our work. We compare our adaptive approaches with these three methods in Table 3.1 and Table 3.2. The outcomes show adaptive Welsch pooling is more desirable than truncated quadratic pooling in top-1 accuracy (while Welsch pooling is worse than truncated quadratic pooling). Our adaptive Huber pooling outperformed truncated quadratic in mean average precision (while Huber pooling is worse than truncated quadratic pooling).

O	Top-1 Accuracy %						Mean Average Precision ×100					
%	M	Q	PH	H	W	TQ	M	Q	PH	H	W	TQ
0	**88.4**	84.7	85.7	86.8	85.5	85.4	95.6	93.8	**95.7**	**95.7**	95.0	93.8
10	79.4	84.3	86.4	87.0	**85.7**	85.5	89.4	94.3	95.7	**95.8**	94.5	94.7
20	76.2	84.8	85.9	**86.9**	85.5	85.5	87.8	94.8	95.6	**95.8**	94.7	95.0
50	72.0	84.0	84.5	84.5	**84.9**	83.9	83.3	93.8	**95.2**	94.8	94.9	94.8
90	29.7	61.7	65.7	65.4	**66.4**	61.8	38.9	76.8	**81.0**	77.9	79.8	76.6

Table 3.1: We compare PointNet with max pooling (M), quadratic (Q), truncated quadratic (TQ) pooling with our adaptive pesudo-Huber (PH), Huber (H) and Welsch (W) pooling. The outliers are presented in training and testing.

O	Top-1 Accuracy %						Mean Average Precision ×100					
%	M	Q	PH	H	W	TQ	M	Q	PH	H	W	TQ
0	**88.4**	84.7	86.6	86.7	85.9	85.4	95.6	93.8	95.0	**95.7**	95.5	93.8
1	32.6	84.9	86.6	**86.8**	85.8	85.3	48.6	93.8	95.0	**95.7**	95.5	93.0
10	6.47	83.9	**86.4**	**86.4**	85.6	85.9	8.20	93.4	94.7	**95.4**	94.8	93.9
20	5.95	79.6	**84.9**	83.8	83.9	**84.9**	7.73	91.9	94.0	**94.6**	93.9	**94.6**
30	5.55	70.9	79.5	78.0	81.8	**83.2**	6.00	87.8	91.4	91.5	92.5	**92.8**
40	5.35	55.3	69.4	68.9	74.1	**75.6**	6.41	77.6	86.5	85.4	88.0	**90.6**
50	4.86	32.9	52.8	58.8	**62.5**	57.9	5.68	62.3	76.5	75.5	80.9	**85.3**
60	4.42	14.5	34.4	41.6	**48.0**	30.6	5.08	39.1	58.7	64.4	66.7	**68.5**
70	4.25	5.03	20.4	21.2	**28.7**	11.9	4.66	22.5	40.3	**49.0**	48.8	47.9
80	3.11	4.10	9.20	6.70	**13.6**	5.11	4.21	10.8	27.2	**35.9**	35.8	26.7
90	3.72	4.06	5.40	4.80	**7.10**	4.22	4.49	8.20	17.2	**18.9**	17.1	9.78

Table 3.2: We compare PointNet with max (M), quadratic (Q), truncated quadratic (TQ) pooling with our adaptive pesudo-Huber (PH), Huber (H) and Welsch (W) pooling. Only test data contains outliers.

3.4.2 ModelNet10

To further analyse the converge of parameter c in our adaptive pooling and show our model works on different datasets, we also conduct experiments on ModelNet10. In

Figure 3.13, we plot the update of c during training. We can observe how quantity of outliers affects the final value of c: if more outliers are presented, there will be more outlier features after the transformation of the CNN layers. Consequently, c shrinks to a smaller value to classify more features as outliers (and give these features less weight). This observation is consistent with our intuition.

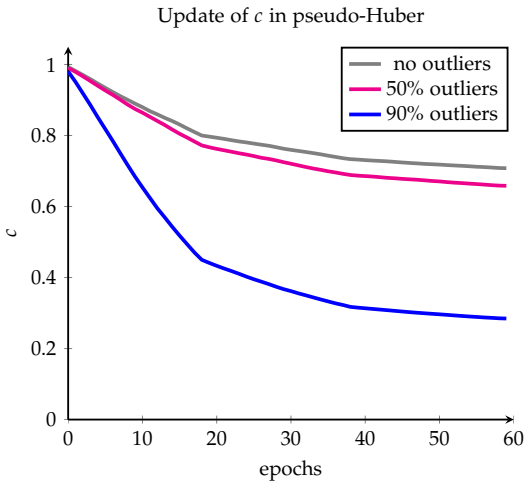

Figure 3.13: Update of parameter c in pseudo-Huber pooling with no outliers , 50% and 90% of outliers presented in training and testing. We trained for 60 epochs.

3.4.3 Runtime Evaluation

Our adaptive pooling adds one trainable parameter c to networks. To further qualify our methods against baselines and native PointNet, we examine the runtime of our method. Inspired by [Gould et al., 2021], we estimate the average training time per point cloud, which includes the forward passing and backward passing. In Table 3.3, we see our methods are around 1.5 to 2 times slower than baselines in forwards passing. In backward passing, runtimes is almost the same, which indicates our gradient implementation is efficient.

Pooling type	Forward (ms)		Backward (ms)	
	baseline	ours	baseline	ours
Pseudo-Huber	**3.75**	6.78	**0.20**	0.25
Huber	**5.63**	9.825	**0.23**	0.15
Welsch	**19.95**	24.23	**0.23**	0.28

Table 3.3: The runtime for adaptive robust pooling on PointNet with ModelNet40. We estimate the average time of forward and backward passing per point cloud. No outlier is presented in training and testing. Our experiments are conducted on single Nvidia RTX2080Ti GPU.

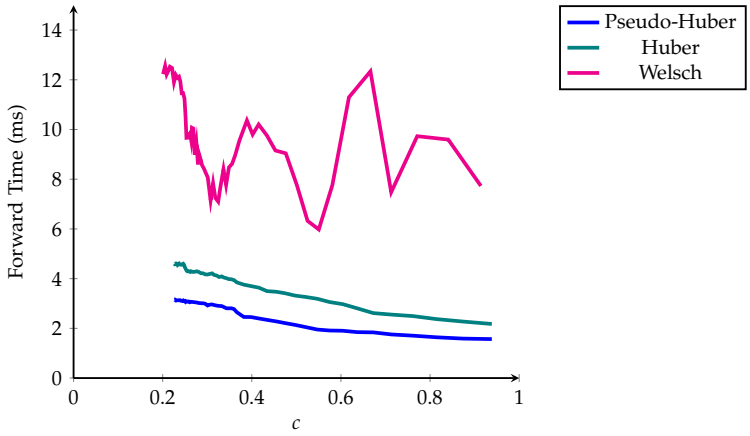

Figure 3.14: The relations between forward runtimes and c in pseudo-Huber, Huber and Welsch pooling. There is no outlier in training and testing.

Figure 3.14 plots the forward runtime and its correlation to parameter c in each epoch. For Huber and pseudo-Huber pooling, we can see smaller c yields higher runtime. When c shrinks, the convexity of Huber and pseudo-Huber decrease (more flat, see Figure 3.1). Consequently, the solving of $\arg\min$ in forward passing takes more time. Welsch function is not convex; thus, the forward runtime is hard to analyse. Because the solving speed of $\arg\min$ in PyTorch is not in the scope of this project, the observation does not indicate our model is less efficient.

3.4.4 General and Adaptive Robust Pooling

Recall section Section 2.2.2, our general and adaptive robust pooling is a superset of pseudo-Huber ($\alpha = 1$), quadratic ($\alpha = 2$) and Welsch ($\alpha = -\infty$) pooling. However, we actively limit the range of α to be $[0, 3]$ to avoid numerical instability. For this range, the general and adaptive robust pooling can generalise quadratic and pseudo-Huber robust pooling. [Gould et al., 2020] has shown quadratic pooling perform poorly with an outlier-contained dataset. In this experiment, we set adaptive and non-adaptive pseudo-Huber as baselines. Moreover, we initialise $\alpha = 1$ and $c = 1$ in general and adaptive pooling; therefore, it is equivalently a pseudo-Huber pooling when the training started.

From Table 3.4, we see the general pooling did not outperform baselines when the outlier rate is low. It performs slightly better than two baselines with 50% and 90% of outliers in top-1 accuracy. The abnormal behaviours with low outlier rate may be caused by numerical instability in complicated gradient formulas (Section 3.3.2). During implementation, we observed there are some unexpected accuracy drops. Unfortunately, we did not resolve this issue in the given time.

Nevertheless, we still have some interesting finds. Figure 3.15 (left and mid) shows the top-1 accuracy and mean average precision of pseudo-Huber, adaptive pseudo-Huber, and general and adaptive pooling. In adaptive and non-adaptive pseudo-Huber, the changes of accuracy and mAP between difference epochs are dra-

O	Top 1 Accuracy %			Mean Average Precision×100		
%	PH	PH(a)	AL	PH	PH(a)	AL
0	**85.7**	**85.7**	82.9	95.6	**95.7**	92.9
10	86.2	**86.4**	85.6	95.1	**95.7**	95.0
20	85.5	**85.9**	85.5	95.0	**95.6**	95.4
50	83.6	84.5	**84.6**	94.9	**95.2**	95.1
90	63.1	65.7	**65.8**	76.8	**81.0**	79.5

Table 3.4: Top-1 accuracy and mean average precision of pseudo-Huber (PH), adaptive pseudo-Huber (PH(a)), and general and adaptive robust pooling (AL). The experiments was conducted in ModelNet40. Outliers are carried in both training and testing.

matic. In general and adaptive pooling, this change is minimal. In Figure 3.15 (right), we see α has decrease from 1 to 0.3. As α decreases, the function applies less penalties on outliers (Figure 2.6 in Section 2.2.1). Consequently, outlier features have less influence on model behaviours.

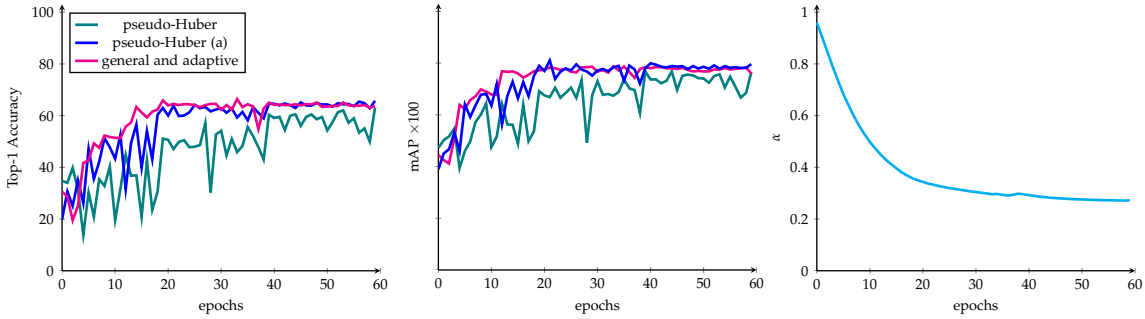

Figure 3.15: **Left**: Top-1 accuracy of pseudo-Huber , adaptive pseudo-Huber (pseudo-Huber (a)), and general and adaptive robust pooling over 60 epochs. **Mid:** mean average precision for these three pooling. **Right:** the update of α in general and adaptive robust pooling. We tested on ModelNet40 with 90% of outliers in training and testing.

3.5 Image Classification

In addition to the point cloud classification task, we investigate our adaptive robust pooling on image classification. Recalling from Section 2.2.1, some prevalent CNN models have global max pooling embedded as a connector between convolutional layers and fully connected layers. ResNet-18 [He et al., 2016] is one of these CNNs. In this work, we take ResNet-18 as our base model and substitute its global max pooling with adaptive robust pooling. we highlighted our modification to the network architecture in Table 3.5 . For comparison, we apply non-adaptive robust pooling from [Gould et al., 2020] as our **baselines**. The baselines have constant threshold $c = 1$, and our adaptive pooling initialise $c = 1$.

layer name	output size	ResNet-18
conv1	112×112×64	7×7, 64, stride 2
conv2_x	56×56×64	3×3 max pooling, stride 2 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$
conv3_x	28×28×128	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$
conv4_x	14×14×256	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$
conv5_x	7×7×512	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$
(adaptive) robust pooling	512	7×7 global robust pooling
fully connected	200	512×200 full connections
softmax	200	

Table 3.5: ResNet-18 architecture [He et al., 2016] with our modification: we replace global max pooling with our adaptive robust pooling. The fully connected layers are adjusted for 200-class dataset.

Figure 3.16: Images from Tiny ImageNet [Wu et al., 2017] with 0 %, 10% , 50% and 90% outliers.

We conduct our experiments on a subset of ImageNet named Tiny ImageNet [Wu et al., 2017], which includes 200 classes, and each class has 500 64 × 64 images. For data preprocessing, we resize images to 256 × 256 and employ a centre crop of 224 × 224. Under such settings, the global pooling is performed over 7 × 7 feature spaces. Moreover, we add noise to images for robustness evaluation: different percentage of pixels are selected and replaced with random float in the range [0, 1] (Figure 3.16). Outliers are carried in training and testing images.

We observed adaptive robust pooling is very sensitive to a large learning rate. Therefore, we apply a fixed learning rate of 0.001 and train for 150 epochs. Such a setting is not practical if we intend to compete with state-of-the-art results. But we set it for analysis purpose. Under such a setting, we still encounter exploding gradients in Huber and Welsch pooling. Therefore, our experiments are conducted with pseudo-Huber pooling only.

O	Top-1 Accuracy %		
%	Avg	PH	PH(a)
0	**47.96**	46.41	45.88
10	**43.08**	42.04	41.72
20	39.93	**40.07**	39.55
50	**33.96**	33.71	33.45
90	**12.57**	12.21	12.35

Table 3.6: Experiments on Tiny ImageNet with different outliers. Avg represents original ResNet-18 with global average pooling; PH is ResNet-18 with pseudo-Huber pooling; PH(a) is ResNet-18 with adaptive pseudo-Huber pooling. We report Top-1 accuracy with different outlier rates. All models train from scratch.

We see the contrast conclusion from point cloud classification: there is an accuracy decrease in adaptive and non-adaptive robust pooling cases. Unfortunately, we did not find the exact reason. We propose two possible explanations:

- One hypothesis is the outliers are being eliminated in the convolution layers of ResNet-18; hence the robust pooling is not playing any roles. And the performance decrease is caused by incorrectly classifying some features as outliers.
- Another possibility is the information of point clouds is more relative to each point's position; nevertheless, information in images is mostly represented by a group of neighbouring pixels. Future work can investigate whether robust pooling improves the robustness with other types of outliers.

3.6 Summary

In this chapter, we reviewed robust pooling and introduced its two-layer adaptive variants. We then formulate a state-of-the-art loss function as a pooling layer. On the one hand, an adaptive c in robust pooling increases model performance constantly and significantly in point cloud classification. Importantly, the performance gain merely costs a minor runtime increase. We also confirmed the converge of c meets our expectations.

On the other hand, our pooling variant of the loss function from [Barron, 2019] is not entirely successful: the implementation is verified to be correct, but we fail to see improvement on point cloud classification. Moreover, performing robust pooling and its adaptive versions on image tasks shows no positive effect.

Adaptive Feature Projections

In this chapter, we first revisit the work of [Gould et al., 2020] on ball and sphere projections. We then describe the shortcoming of current methods and how we develop a more flexible approach in Section 4.1. Moreover, we acquire the gradient formulas for our methods in Section 4.2. Last, we conduct experiments on several CNN models for image classification in Section 4.3.

Euclidean projection (also called L_2 normalization), as a type of feature projections, transforms data from high-dimension to lower dimension. This reduction aims to capture essential properties of data in fewer dimension space [Murphy, 2012].

We first briefly present the deviation of sphere and ball projections in [Gould et al., 2020]. Euclidean projection onto L_2 norm takes data and converts it into a unit vector:

$$y = \frac{1}{||x||_2} x \tag{4.1}$$

where

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

We can write Equation (4.1) as a single constrained bi-level optimisation:

$$\begin{aligned} y \in \quad & \arg\min_{u \in \mathbb{R}^n} \quad \frac{1}{2}\|u - x\|_2^2 \\ & \text{subject to} \quad \|u\|_2 = 1. \end{aligned} \tag{4.2}$$

Euclidean projection onto other types of norm may be used for regularization to improve network performance [Oymak, 2018]. Equation (4.2) can be generalised to to L_p norm:

$$\begin{aligned} y_p \in \quad & \arg\min_{u \in \mathbb{R}^n} \quad \frac{1}{2}\|u - x\|_2^2 \\ & \text{subject to} \quad \|u\|_p = 1 \end{aligned} \tag{4.3}$$

where $||u||_p$ is the projection onto L_p sphere

$$||x||_p = \sum_{i=1}^{n} (|x_i|^p)^{\frac{1}{p}}.$$

Relaxing the constraint in Equation (4.3), we have L_p ball projections:

$$y_p \in \begin{array}{ll} \arg\min_{u \in \mathbb{R}^n} & \frac{1}{2}\|u - x\|_2^2 \\ \text{subject to} & \|u\|_p \leq 1. \end{array} \tag{4.4}$$

For $p = 1, \infty$, L_1 and L_∞ norm are defined as:

$$||x||_1 = \sum_{i=1}^{n}(|x_i|)$$
$$||x||_\infty = max_{i=1}^{n}(|x_i|)$$

Before proceeding to our approach, we first get a sense of how these projections look like from some plots in [Gould et al., 2020]. Figure 4.1 is the Euclidean projections of $x \in \mathbb{R}^2$ on L_p norm spheres. From left to right, it corresponds to L_2, L_1 and L_∞ unit sphere projections. y_p represents the projected point. We see the projection on L_2 is smooth and have a unique solution; projections on L_1 and L_∞ are not smooth and have isolated solutions [Gould et al., 2020].

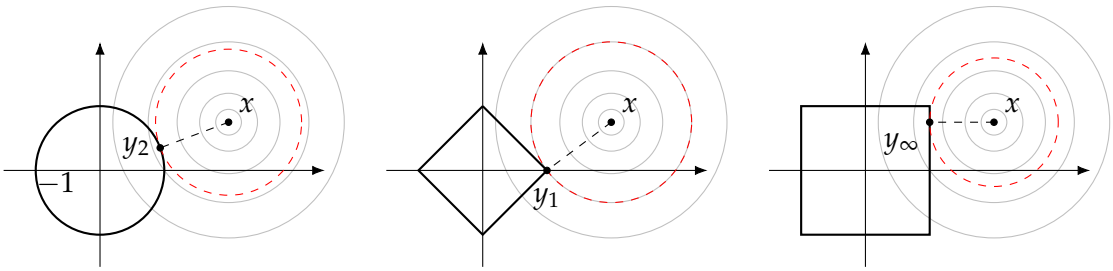

Figure 4.1: From left to right: Euclidean projection onto unit L_2, L_1 and L_∞ spheres ($r = 1$). [Gould et al., 2020]

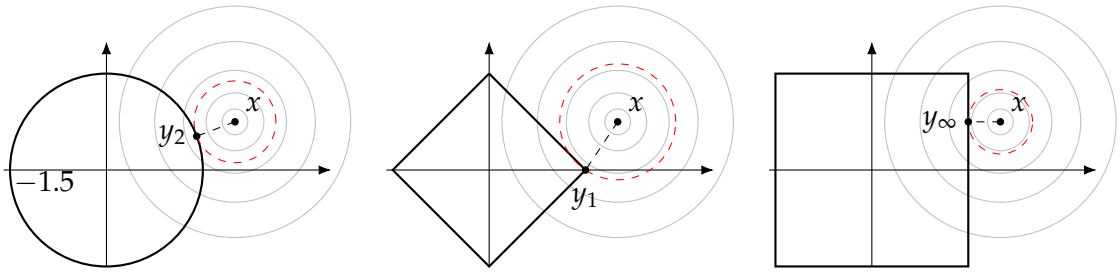

Figure 4.2: From left to right: Euclidean projection onto L_2, L_1 and L_∞ spheres with radius $r = 1.5$. [Gould et al., 2020]

[Duchi et al., 2008; Gould et al., 2020] provide efficient solvers for L_1 and L_∞ projections. With DDN propositions, we are able to compute the gradient for $Dy(x)$ for L_1, L_2 and L_∞ ball and sphere projections. Therefore, we can embed these projections as declarative nodes in networks. Sphere and ball projections are classified as equality constrained (Equation (2.19)) and inequality constrained (Equation (2.21)) declarative nodes respectively.

Experiments have confirmed projection layers raise prediction confidence on image tasks. In ResNet-18, if we insert a batchnorm and ball or sphere projections layers before the last fully connected layers, there is a significant gain on mean average precision [Gould et al., 2020]. Their experiments will serve as baselines in Section 4.3.

4.1 Definition

In the experiments from [Gould et al., 2020], the features were pre-scaled before feeding into projection layers. The scaling factors are 250, 15 and 3 for L_1, L_2 and L_∞. These scaling factors correspond to the radius r in the constraint of projections:

$$\begin{aligned} y_p \in \quad & \arg\min_{u \in \mathbb{R}^n} \quad \tfrac{1}{2}\|u - x\|_2^2 \\ & \text{subject to} \quad ||u||_p = \begin{cases} 250 & \text{if } p = 1 \\ 15 & \text{if } p = 2 \\ 3 & \text{if } p = \infty \end{cases} \end{aligned}$$

where constant r is approximated as $\frac{2}{3}$ of the median of the features $||f_i||_p$, which guarantees projections cover the majority of features but not too aggressive [Gould et al., 2020]. Nevertheless, this approach is not flexible for broader applications. If we apply projections to new models or new images, we must manually tune scaling factor r before training starts.

We examine how different radius r affect the projected y_p in Figure 4.2. Instead of projecting on unit spheres, x in Figure 4.2 are projected on L_2, L_1 and l_∞ spheres with radius $r = 1.5$. We can see the properties of L_2, L_1 and l_∞ projections still preserve; however, we get different solutions for y_p. Applying the similar methodology for adaptive robust pooling, we configure r as a positive parameter. In sphere projections:

$$\begin{aligned} y_p \in \quad & \arg\min_{u \in \mathbb{R}^n} \quad \tfrac{1}{2}\|u - x\|_2^2 \\ & \text{subject to} \quad ||u||_p = r \\ & \qquad\qquad\quad r > 0 \end{aligned} \tag{4.5}$$

We notice the second constraint cause similar difficulties as in Section 3.1.1: Equation (4.5) is a declarative node with multiple constraints (DDN with multiple constraints is well-studied in [Wang, 2020]), and they require more complicated gradient computation. We enforce the second constraint by applying the affine softplus transformation in Equation (3.9) to r. Our **adaptive sphere projections** are a composition

of two layers:

$$\begin{aligned}
&\text{Input:} && x \\
&\text{Parameter:} && r^* \\
&\text{Transformation Layer:} && r = f(r^*) \\
& && \quad \text{where } f \text{ is the affine softplus defined in Equation (3.9)} \\
&\text{Declarative Layer:} && y_p \in \arg\min_{u \in \mathbb{R}^n} \tfrac{1}{2}\|u - x\|_2^2 \\
& && \quad \text{subject to } ||u||_p = r,\ p = 1, 2 \text{ or } \infty \\
&\text{Forward Output:} && y_p \\
&\text{Backward Output:} && \mathrm{D}_X y_p(x, r^*) \\
& && \mathrm{D}_R y_p(x, r^*)
\end{aligned} \tag{4.6}$$

For the **adaptive ball projections**, we relax the constraint:

$$\begin{aligned}
&\text{Input:} && x \\
&\text{Parameter:} && r^* \\
&\text{Transformation Layer:} && r = f(r^*) \\
& && \quad \text{where } f \text{ is the affine softplus defined in Equation (3.9)} \\
&\text{Declarative Layer:} && y_p \in \arg\min_{u \in \mathbb{R}^n} \tfrac{1}{2}\|u - x\|_2^2 \\
& && \quad \text{subject to} ||u||_p \leq r,\ p = 1, 2 \text{ or } \infty \\
&\text{Forward Output:} && y_p \\
&\text{Backward Output:} && \mathrm{D}_X y_p(x, r^*) \\
& && \mathrm{D}_R y_p(x, r^*)
\end{aligned} \tag{4.7}$$

4.2 Gradients

The derivative calculations in Equation (4.6) and Equation (4.7) are similar to adaptive robust pooling in Section 3.3: Pytorch AutoGrad [Paszke et al., 2017] can take care of the gradients in transformation layers. y_p is an implicit function of x and r; we represent the relation as $y(x, r)$; $h(u, r)$ is a function represents the constraint between u and r. $f(x, r, y)$ represents the arg min objective. The gradient $\mathrm{D}_X y(x, r)$ is identical to $\mathrm{D}y(x)$ in non-adaptive projections in [Gould et al., 2020]. Therefore, we are only required to calculate $\mathrm{D}_R y(x, r)$ in the declarative layer. The complete deviations with steps are given in Appendix A.2.

4.2.1 Adaptive Sphere Projections

We first present gradient via closed-formed L_2 projection. Defining adaptive L_2 projection explicitly:

$$y = \frac{r}{||x||_2} x \tag{4.8}$$

and the gradient $\mathrm{D}_R y(x,r)$ is trivial to compute:

$$\mathrm{D}_R y(x,r) = \frac{x}{\|x\|_2}. \tag{4.9}$$

We calculate $\mathrm{D}_R y(x,r)$ as an equality constrained declarative node by applying Equation (2.20):

$$\begin{aligned}
&\text{arg min objective:} \\
&\qquad f(x,r,y) &&= \tfrac{1}{2}\|u-x\|_2^2 \\
&\text{Constraint:} \\
&\qquad h(r,u) &&= \|u\|_2 - r \\
&\qquad &&= \sqrt{\textstyle\sum_{i=1}^n u_i^2} - r \\
&\text{Gradients:} \\
&\qquad \mathrm{D}_X y(x,r) &&= \frac{1}{\|x\|_2}\left(I - yy^T\right) \\
&\qquad \mathrm{D}_R y(x,r) &&= \frac{y}{\|y\|_2} \\
&\qquad &&= \frac{\frac{r}{\|x\|_2}x}{\|\frac{r}{\|x\|_2}x\|_2} \\
&\qquad &&= \frac{x}{\|x\|_2}.
\end{aligned} \tag{4.10}$$

Comparing Equation (4.9) and Equation (4.10), we obtained identical formulas, which verifies the declarative adaptive L_2 sphere projection is equivalent to its imperative form. L_1 and L_∞ projections do not have closed-form solutions:

- L_1 projections:

$$\begin{aligned}
&\text{arg min objective:} \\
&\qquad f(x,r,y) &&= \tfrac{1}{2}\|u-x\|_2^2 \\
&\text{Constraint:} \\
&\qquad h(r,y) &&= \|u\|_1 - r \\
&\qquad &&= \textstyle\sum_{i=1}^n |u_i| - r \\
&\text{Gradients:} \\
&\qquad \mathrm{D}_X y(x,r) &&= I - \frac{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y) \mathrm{D}_Y h(r,y)^T} \\
&\qquad \mathrm{D}_R y(x,r) &&= \frac{\mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}
\end{aligned} \tag{4.11}$$

- L_∞ projections:

  arg min objective:

  $$f(x,r,y) = \tfrac{1}{2}\|u-x\|_2^2$$

  Constraint:

  $$\begin{aligned} h(r,y) &= \|u\|_\infty - r \\ &= \max_i\{|u_i|\} - r \end{aligned}$$

  Gradients:

  $$\begin{aligned} \mathrm{D}_X y(x,r) &= I - \frac{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y)\mathrm{D}_Y h(r,y)^T} \\ \mathrm{D}_R y(x,r) &= \mathbf{vec}\,\{z_i\} \\ \text{where } z_i &= \begin{cases} 0 & \text{if } \mathrm{D}_Y h(r,y)_i = 0 \\ \mathbf{sign}\,(\mathrm{D}_Y h(r,y)_i) & \text{otherwise} \end{cases} \end{aligned} \tag{4.12}$$

4.2.2 Adaptive Ball Projections

The ball projections are classified as inequality constrained declarative nodes Equation (2.21). Their gradient deviations are very similar to sphere projections. We can summarise into two situations:

1. When data projects on sphere, the constraint h is active

   $$\begin{aligned} h(r,y) &= \|u\|_p - r \\ &= 0. \end{aligned} \tag{4.13}$$

   the gradients for L_2, L_1 and L_n ball projections are identical to sphere projections in Equation (4.10), Equation (4.11) and Equation (4.12) .

2. When data projects inside of sphere, the constraint h is no active:

   $$\begin{aligned} h(r,y) &= \|u\|_p - r \\ &\neq 0. \end{aligned} \tag{4.14}$$

   we have

   $$\begin{aligned} \mathrm{D}_X y(x,r) &= I \\ \mathrm{D}_R y(x,r) &= 0 \end{aligned} \tag{4.15}$$

We utilise chain rule for the gradient in transformation layer:

$$\mathrm{D}_{R^*} y(x,,r^*) = \mathrm{D}_X y(x,r) \cdot \mathrm{D}_{C^*} f_1(r^*) \tag{4.16}$$

where $f_1(r^*)$ is the softplus function defined in Equation (3.9). And the backpropagation in $f_1(r^*)$ uses the Pytorch AutoGrad library. The complete vectorised Python PyTorch [Paszke et al., 2017] implementation of adaptive L_1, L_2 and l_∞ ball, sphere projections can be found at: https://github.com/WenboDu1228/ddn_pooling_and_

`projections`. We have verified the correctness of gradient formulas with the Python AutoGrad GradCheck library and some examples.

4.3 Image Classification

The effect of the sphere and ball projections layer was evaluated on ResNet-18 [He et al., 2016] with ImageNet 2012 in [Gould et al., 2020]: if we insert a batchnorm and a projection layer before fully connected layers, there is a notable gain in mean average precision. In this work, we treat their experiments as our **baselines**. Due to the limit of computation resource, we experiment on two smaller datasets:

1. **CIFAR10** [Krizhevsky, 2009] is a 10 classes dataset with 6000 64×64 images in each category. It is one of the most used datasets in computer vision research.
2. **Imagewoof** [Husain, 2019] is a 10 classes subset of ImageNet. The images are not easy to classify since they are all dog breeds. Figure 4.3 are some images from different classes.

Figure 4.3: Three classes of dog breeds in Imagewoof dataset [Husain, 2019].

Other settings are consistent with the baselines: we resize images to 256×256 and apply a central crop of 224×224 to training images. The optimisation method is stochastic gradient descent with 90 epochs. The learning rate is initialised as 0.1 and decrease by 10 times every 30 epochs. We set the batch size as 128. We report top-1 accuracy and mean average precision on the validation set.

We substitute sphere and ball projections with our adaptive projections. In Table 4.1, we highlighted such modifications on ResNet-18. For baselines, we set constant r to be 250 and 15 for L_1 and L_2 projections according to [Gould et al., 2020]. For adaptive projections, we initialise r as 250 and 15 for L_1 and L_2 projections. Because we witnessed radius r in adaptive L_∞ projections moves towards 0 quickly and produces abnormal output (although the gradient implementation passes Pytorch GradCheck), we did not include the results here. Our experiments are comprised of two parts:

1. We experiment on ResNet-18 as suggested in [Gould et al., 2020] on two datasets and then compare our adaptive projections with baselines.
2. We insert projection layers into other networks and observe if described methods have performance gain in general CNN architectures.

layer name	output size	ResNet-18
conv1	112×112×64	7×7, 64, stride 2
conv2_x	56×56×64	3×3 max pooling, stride 2 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$
conv3_x	28×28×128	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$
conv4_x	14×14×256	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$
conv5_x	7×7×512	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$
average pool	512	7×7 average pooling
batchnorm	512	
(adaptive) projection	512	
fully connected	10	512×10 full connections
softmax	10	

Table 4.1: ResNet-18 architecture with our modifications: we insert a batchnorm and adaptive (our) or non-adaptive (baselines) projections before fully connected layers. The modifications are highlighted in red. The fully connected layers are adjusted for 10-class dataset [He et al., 2016].

4.3.1 ResNet-18

Figure 4.4 includes our experiments on ResNet-18. Comparing projection-inserted ResNet-18 and original ResNet-18, we find projections improve top-1 accuracy moderately in Imagewoof and slightly in CIFAR10. The improvement in mean average precision is evident in both datasets. This observation is consistent with ImageNet experiments in [Gould et al., 2020]. Furthermore, comparing radius-fixed projections (baselines) and adaptive projections (ours), we notice our methods increase top-1 accuracy moderately and mean average precision exceptionally. On average, L_2 projections perform better than L_1 projections. We are positive to say the performance gain in adaptive projections is consistent. The complete data is summarised in Appendix B.2.

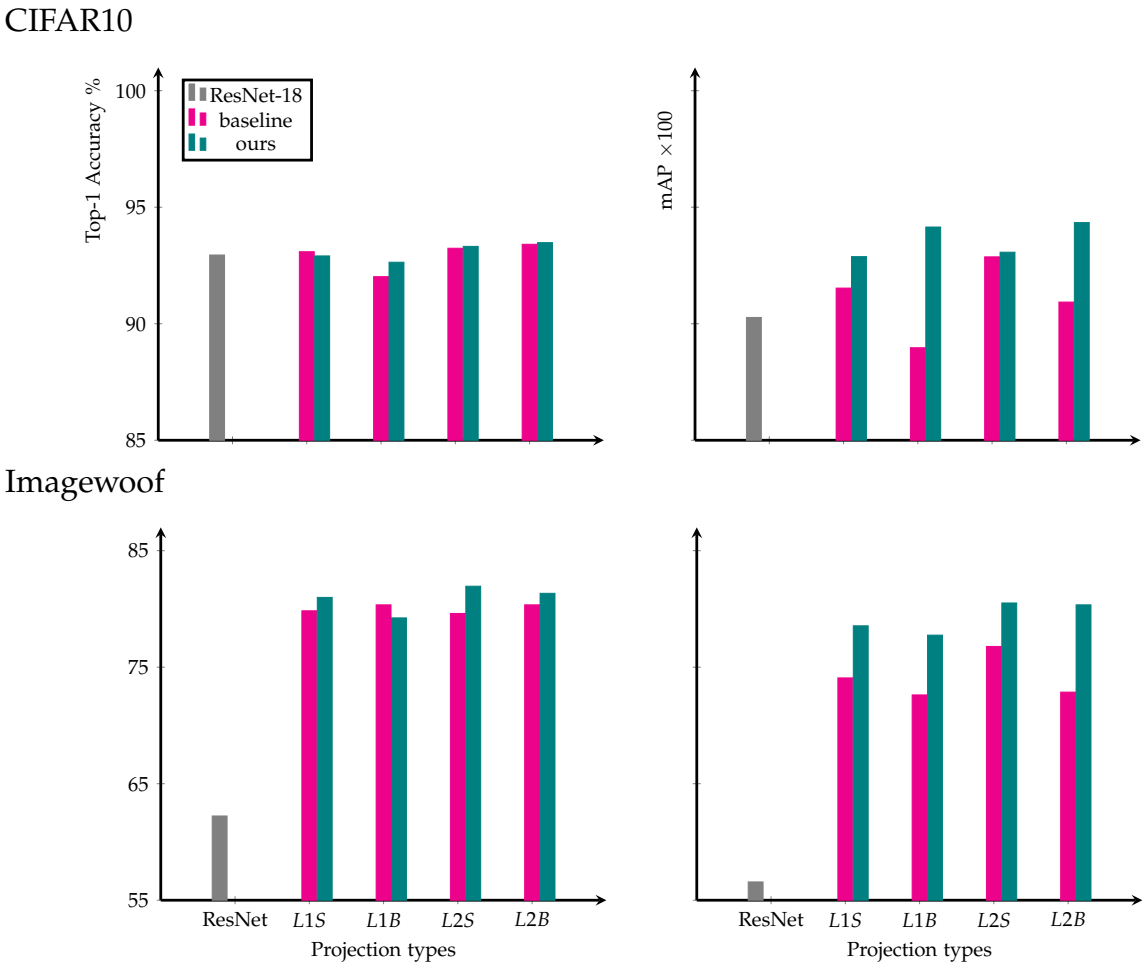

Figure 4.4: Experiments ResNet-18. **Top:** results on CIFAR10. **Bottom:** results on Imagewoof. **Left:** top-1 accuracy. **Right:** mean average precision (mAP). We report results on Resnet-18 without projection layers in the left gray bar. The other settings are projection-inserted ResNet-18. *L1S*, *L1B*, *L2S* and *L2B* represent L_1 sphere, L_1 ball, L_2 sphere and L_2 ball projections. All models train from scratch. Outcomes for non-adaptive and adaptive ones are shown in magenta and teal bars respectively. The y axis in top and bottom figures do not have same starting numbers.

4.3.2 GoogLeNet and DenseNet-121

[Gould et al., 2020] and our work in the prior section have shown inserting projections in ResNet-18 produces favourable results. Likewise, we desire to demonstrate such an impact applies to general CNN architectures. Therefore, we additionally experiments with GoogLeNet [Szegedy et al., 2015] and DenseNet-121 [Huang et al., 2017]. Similar to ResNet-18, we prepend fully connected layers with a batchnorm and projection layer.

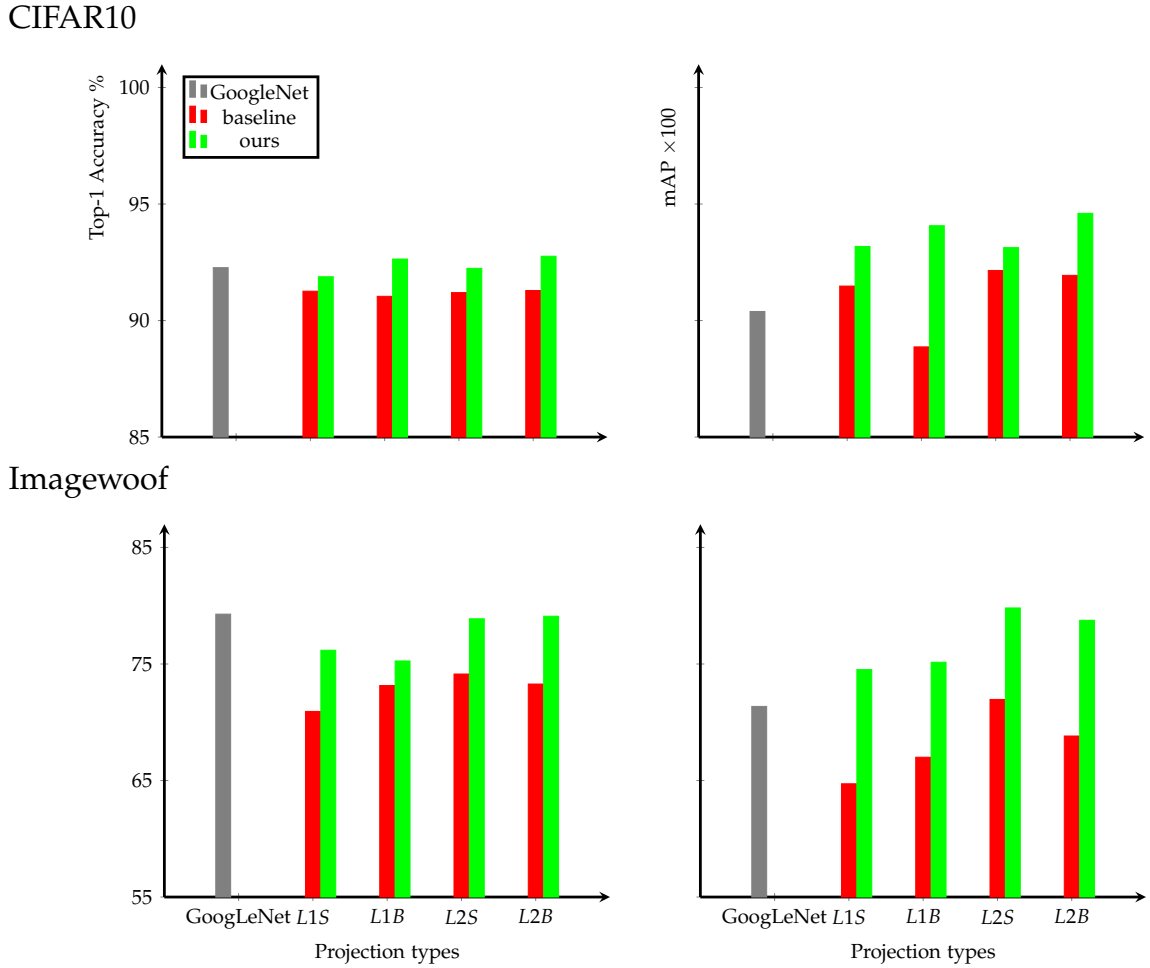

Figure 4.5: Experiment with GoogLeNet. We report results on GoogLeNet without projection layers in the left gray bar. Other bars are results projection-inserted GoogLeNet (consistent with Figure 4.4).

The results are plotted in Figure 4.5 and Figure 4.6. In majority cases, we see non-adaptive projections produce poorer results than the native GoogLeNet and DenseNet-121, which does not agree with the observations in ResNet-18. Contrarily, our adaptive variants constantly exceed original CNN models. The top-1 accuracy increases slightly, but there is a considerable improvement in mean average precision. The specific data for these two figures is included in Appendix B.2.

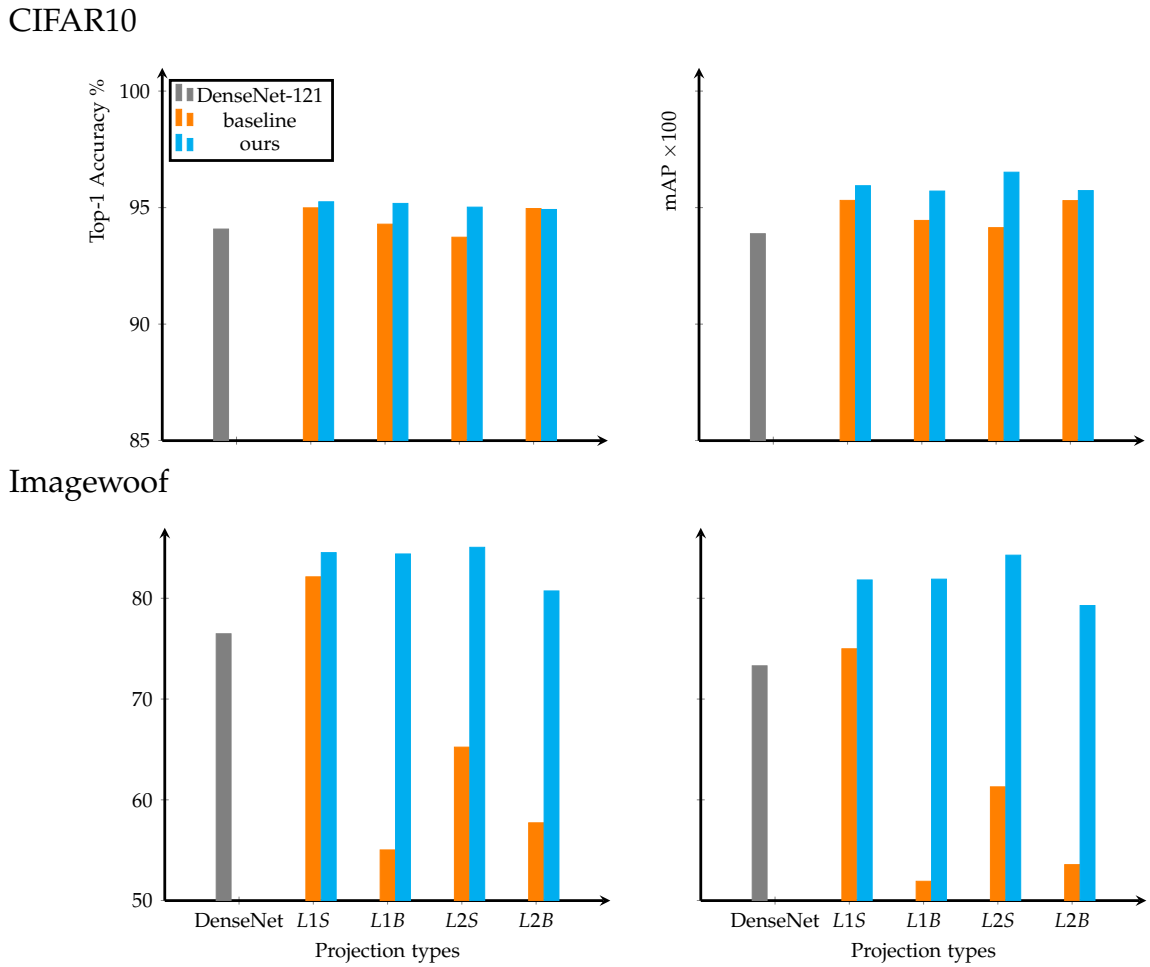

Figure 4.6: Experiment with DenseNet-121. We report results on DenseNet-121 without projection layers in the left gray bar. Other bars are projection-inserted DenseNet-121 (consistent with Figure 4.4).

In non-adaptive projections, the constant r was pre-determined for ResNet-18; hence, they may not be exemplary for other networks. It explains why we observed original GoogLeNet and DenseNet-121 produce better results than their non-adaptive projection-inserted versions. The comparison between adaptive projections and original CNN models has further validated the benefit of our methods: In non-adaptive configurations, we have to determine values of r carefully before training start. Otherwise, projection layers may negatively affect training. Nevertheless, this effect has been minimised in our adaptive versions since networks are capable of learning a proper r.

GoogLeNet, DenseNet-121 and ResNet-18 all contain one fully connected layers at the end. We further conducted experiments on VGG-11 [Simonyan and Zisserman, 2014] and AlexNet [Krizhevsky et al., 2012] in Appendix B.2. Both networks contain multiple fully connected and non-linear layers at the end. We attempted to

insert a batchnorm and projection layer between different fully connected layers. The results indicates adaptive projections outperform original CNN models that do not contain projection layers. However, they do not further improve from non-adaptive projections. We purpose two possible reasons for these behaviours:

1. The positive impact of adaptive projection layers are limited to models with one fully connected layers.

2. The appropriate r for these two models may be very far from our preceding initial settings (250 for L_1 and 15 for L_2). Consequently, r did not converge in given epochs.

4.3.3 Runtime Evaluation

Similar to adaptive robust pooling, our adaptive projections add one parameter to networks. In Table 4.2, we reports runtime per image with ResNet-18 on CIFAR10 dataset. We observe our adaptive projection did not increase runtime. Therefore, our implementation of adaptive projections is efficient.

Model	Runtime (ms)	
	baseline	ours
ResNet-18	1.110	-
ResNet-18 L_1 Sphere	1.170	**1.166**
ResNet-18 L_2 Sphere	**1.634**	1.656
ResNet-18 L_1 Ball	1.164	**1.156**
ResNet-18 L_2 Ball	**1.151**	1.174

Table 4.2: Runtime comparsion between ResNet-18 with non-adaptive projections (baselines) and adaptive projections (ours). We estimate runtime per image on CIFAR10 dataset. ResNet-18 entries records runtime for ResNet-18 without projection layers. All experiments are conducted on single Nvidia RTX2080Ti GPU.

4.4 Summary

We have introduced two-layer adaptive variants of sphere and ball projections. On the one hand, our adaptive features projections produce consistent and exceptional performance improvement in several CNN models with similar architectures as ResNet-18. Importantly, They do so with negligible additional memory and runtime.

On the other hand, the adaptive L_∞ projections did not work in practice. For general CNN models, we did not collect sufficient evidence on the promising effect of projections. In future, experiments on more comprehensive datasets like ImageNet will provide more insight into our algorithms.

Chapter 5

Conclusion

In this work, we investigated two instances of deep declarative networks and proposed their parametric versions. In particular, we introduced a new type of pooling with adjustable robustness.

In Chapter 2, we reviewed some related work and theoretical background. Our adaptive robust pooling is an instance of pooling layers which normal placed in the mid of convolutional neural networks. The backpropagation of adaptive robust pooling and feature projections is very similar to conventional network layers. We then delved into a state-of-the-art loss function that is capable of controlling robustness. Furthermore, we revisited deep declarative networks. In particular, we focused on the forward and backward process of declarative nodes.

In Chapter 3, we proposed a new pooling layer named adaptive robust pooling. We started with the earlier version of robust pooling and discussed its limitations. The main research question is: how we determine the constant c in robust pooling? Our approach is to learn c in end-to-end networks. We then recognised some difficulties of our approach and resolved them with affine transformations. In addition, we introduced another robust pooling with more flexibility and interpretability. We called such pooling general and adaptive robust pooling.

Furthermore, we conducted experiments on point cloud classification and image classification. Our methods improve top-1 accuracy and mean average precision with little extra cost in point cloud classification. However, we did not see the positive impact of robust pooling on image classification tasks.

In Chapter 4, we presented our adaption of prior work on feature projections. We named our methods adaptive feature projections because the scaling factor r is updated via network backpropagation. Next, we obtained gradients of our projections methods using the theorem from deep declarative networks. We selected non-adaptive projections as baselines and validated our adaptive approaches outperform baselines consistently and remarkably. Moreover, we observed the beneficial effects of our projection layers exist in multiple convolutional neural networks. Crucially, the runtime and memory increase of our methods are negligible.

Due to the time scope of the project, there are several ideas that we did not further investigate. These ideas include:

- **Local robust pooling:** our experiments on point clouds and images were all on

global pooling. Compared with global pooling, local pooling is more prevalent in computer vision applications. In future, we can adapt robust pooling as local pooling and research its resistance to outliers.

- **Independent parameters in pooling and projections:** the loss from [Barron, 2019] has independent parameters for each input feature. This advantage did not migrate to our pooling: we have a share value of c and α for all input channels. In future work, we can research on having separated parameters for different input channels. Such a setting can be advantageous when input is a mix of clear and contaminated data. Besides, we can learn parameter from the input: Figure 5.1 presents one of the proposed methods. We attempt to use fully connected layers to approximate the outlier threshold c. Similar ideas apply to feature projections.

- **Applications of general parametric deep declarative networks:** our research concentrated on two instances of deep declarative networks. There are many other instances of deep declarative networks that may extend to parametric settings. Before and concurrent with my project, other work has indicated the positive impact of robust batch normalisation on defending adversarial attacks. In future, an interesting research direction is to introduce learnable parameters in robust batch normalisation.

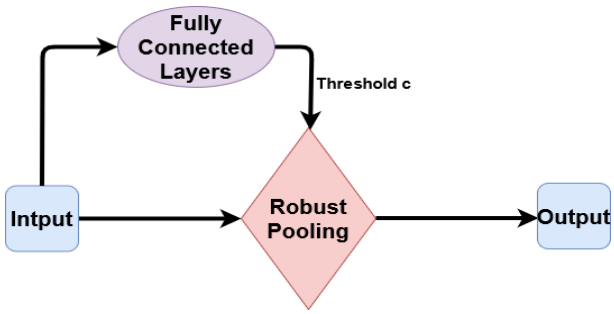

Figure 5.1: One of the proposed architecture. We attempt to estimate the the outlier threshold c with fully connected layers.

In conclusion, deep declarative networks are relative new research areas. We made some insightful study on two instances of deep declarative networks. In future, we hope deep declarative networks help with bridging deep learning with various research problems and industrial challenges.

Bibliography

Agrawal, A.; Barratt, S.; Boyd, S.; Busseti, E.; and Moursi, W. M., 2019. Differentiating through a cone program. Technical report, Stanford University (arXiv:1904.09043). (cited on page 10)

Amos, B. and Kolter, J. Z., 2017. OptNet: Differentiable optimization as a layer in neural networks. In *ICML*. (cited on pages 1 and 10)

Amos, B.; Rodriguez, I. D. J.; Sacks, J.; Boots, B.; and Kolter, J. Z., 2018. Differentiable MPC for end-to-end planning and control. In *NIPS*. (cited on page 10)

Bard, J. F., 1998. *Practical Bilevel Optimization: Algorithms and Applications*. Kluwer Academic Press. (cited on page 12)

Barron, J. T., 2019. A general and adaptive robust loss function. In *CVPR*. (cited on pages xiii, 2, 7, 8, 9, 10, 17, 19, 23, 37, and 52)

Black, M. J. and Ananda, P., 1996. The robust estimation of multiple motions: Parametric and piecewise-smooth flowfields. In *CVIU*. (cited on pages 7 and 8)

Boyd, S. P. and Vandenberghe, L., 2004. *Convex Optimization*. Cambridge. (cited on page 1)

Charbonnier, P.; Blanc-Feraud, L.; Aubert, G.; and Barlaud, M., 1997. Deterministic edge-preserving regularization in computed imaging. *IEEE Transactions on Image Processing*, 6, 2 (1997), 298–311. doi:10.1109/83.551699. (cited on pages 6 and 17)

Chen, T. Q.; Rubanova, Y.; Bettencourt, J.; and Duvenaud, D. K., 2018. Neural ordinary differential equations. In *NIPS*. (cited on pages 1 and 10)

Cook, A., 2017. (cited on pages xiii and 5)

Cranmer, M.; Greydanus, S.; Hoyer, S.; Battaglia, P.; Spergel, D.; and Ho, S., 2020. Lagrangian neural networks. In *ICLR*. (cited on page 10)

Dennis, J. E. and Welsch, R. E., 1978. Techniques for nonlinear least squares and robust regression. *Communications in Statistics - Simulation and Computation*, 7, 4 (1978), 345–359. (cited on pages 6 and 18)

Dontchev, A. L. and Rockafellar, R. T., 2014. *Implicit Functionsand Solution Mappings: A View from Variational Analysis*. Springer-Verlag, 2nd edition. (cited on page 13)

Duchi, J.; Shalev-Shwartz, S.; Singer, Y.; and Chandra, T., 2008. Efficient projections onto the $l1$-ball for learning in high dimensions. In *ICML*, 272–279. (cited on page 40)

Geman, S. and McClure, D. E., 1985. Bayesian image anal-ysis: An application to single photon emission tomography. *Proceedings of the American Statistical Association*, (1985). (cited on pages 7 and 10)

Godard, C.; Aodha, O. M.; and Brostow, G. J., 2017. Unsupervised monocular depth estimation with left-right consistency. In *CVPR*. (cited on page 10)

Gould, S.; Fernando, B.; Cherian, A.; Anderson, P.; Santa Cruz, R.; and Guo, E., 2016. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. Technical report, Australian National University (arXiv:1607.05447). (cited on pages 10 and 15)

Gould, S.; Hartley, R.; and Campbell, D., 2020. Deep declarative networks: A new hope. In *CVPR*. (cited on pages xiii, xiv, xviii, 1, 3, 6, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 25, 28, 29, 30, 31, 32, 34, 35, 39, 40, 41, 42, 45, 46, 48, and 61)

Gould, S.; Hartley, R.; and Campbell, D. J., 2021. Deep declarative networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2021), 1–1. doi:10.1109/TPAMI.2021.3059462. (cited on page 33)

Greydanus, S.; Dzamba, M.; and Yosinski, J., 2019. Hamiltonian neural networks. In *NeurIPS*. (cited on page 10)

Grubbs, F. E., 1969. Procedures for detecting outlying observations in samples. *Technometrics*, 11, 1 (1969), 1–21. doi:10.1080/00401706.1969.10490657. (cited on page 6)

Hastie, T.; Tibshirani, R.; and Wainwrig, M., 2015. *Statistical Learning with Sparsity: The Lasso and Generalizations*. CPC Press. (cited on page 6)

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2016. Deep residual learning for image recognition. In *CVPR*. (cited on pages xvii, 5, 35, 36, 45, and 46)

Huang, G.; Liu, Z.; van der Maaten, L.; and Weinberger, K. Q., 2017. Densely connected convolutional networks. In *CVPR*. (cited on pages 5 and 48)

Huber, P. J., 1964. Robust estimation of a location parameter. In *Annals*. (cited on pages 6 and 17)

Huber, P. J., 1981. *Robust Statistics*. John Wiley. (cited on page 6)

Husain, H., 2019. Imagenette. https://github.com/fastai/imagenette. (cited on pages xiv and 45)

Iandola, F. N.; Han, S.; Moskewicz, M. W.; Ashraf, K.; and William J. Dally, K. K., 2017. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. In *ICLR*. (cited on page 5)

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *PMLR*. (cited on page 5)

Kingma, D. P. and Welling, M., 2014. Auto-encoding variational bayes. In *ICLR*. (cited on page 10)

Krizhevsky, A., 2009. Learning multiple layers of features from tiny images. Technical report, University of Toronto. (cited on page 45)

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E., 2012. Imagenet classification with deep convolutional neural networks. In *NeurIPS*. (cited on page 49)

Leclerc, Y. G., 1989. Constructing simple stable descriptions forimage partitioning. In *IJCV*. (cited on page 6)

Lin, M.; Chen, Q.; and Yan, S., 2014. Network in network. In *ICLR*. (cited on page 5)

Liu, Z.; Luo, P.; Wang, X.; and Tang, X., 2015. Deep learning face attributes in the wild. In *ICCV*. (cited on page 10)

Lutter, M.; Ritter, C.; and Peters, J., 2019. Deep lagrangian networks: Using physics as model prior for deep learning. In *ICLR*. (cited on page 10)

Maddala, G. S., 1992. *Introduction to Econometrics*. Macmillan Pub. Co. (cited on page 6)

Murphy, K. P., 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press. (cited on page 39)

Oymak, S., 2018. Learning compact neural networks with regularization. In *ICML*, 3963–3972. (cited on page 39)

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A., 2017. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*. (cited on pages 25, 27, 42, and 44)

Qi, C. R.; Su, H.; Mo, K.; and Guibas, L. J., 2016. PointNet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*. (cited on pages xiv and 28)

Saha, S., 2018. A comprehensive guide to convolutional neural networks. (cited on pages xiii and 4)

Shah, S. A. and Koltu, V., 2017. Robust continuous clustering. In *PNAS*. (cited on page 10)

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1811.00982*, (2014). (cited on page 49)

Steven, K. and Harold, P., 2013. *The Implicit Function Theorem*. Modern Birkhauser Classics. Birkhauser. ISBN. (cited on page 11)

Szegedy, C.; Wei Liu, Y. J.; Sermanet, P.; Scott Reed, D. A.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A., 2015. Going deeper with convolutions. In *CVPR*. (cited on pages 5 and 48)

von Stackelberg, H.; Bazin, D.; Urch, L.; and Hill, R. R., 2011. *Market structure and equilibrium*. Springer. (cited on page 12)

Wang, P.-W.; Donti, P. L.; Wilder, B.; and Kolter, Z., 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *ICML*. (cited on pages 1 and 10)

Wang, S., 2020. Multiple constraints and non-regular solution in deep declarative network. Technical report, The Australian National University. (cited on pages 16 and 41)

Wu, J.; Zhang, Q.; and Xu, G., 2017. Tiny imagenet challenge. (cited on pages xiv and 36)

Wu, Z.; Song, S.; Khosla, A.; Yu, F.; Zhang, L.; Tang, X.; and Xiao, J., 2015. 3d shapenets: A deep representation for volumetric shapes. In *CVPR*. (cited on page 28)

Xu, B.; Wang, N.; Chen, T.; and Li, M., 2015. Empirical evaluation of rectified activations in convolutional network. In *arXiv*. (cited on page 5)

Yamashita, R.; Nishio, M.; Do, R. K. G.; and Togashi, K., 2018. Convolutional neural networks: an overview and application in radiology. *sights into Imaging*, (2018), 1. (cited on page 3)

Yu, D.; Wang, H.; Chen, P.; and Wei, Z., 2014. Mixed pooling for convolutional neural networks. In *RSKT*, 364–375. (cited on page 4)

Zach, C., 2014. Robust bundle adjustment revisited. In *ECCV*. (cited on page 10)

Zhou, B.; Aditya Khosla, A. L.; Oliva, A.; and Torralba, A., 2016a. Learning deep features for discriminative localization. In *CVPR*. (cited on pages 4 and 5)

Zhou, Q.-Y.; Park, J.; and Koltu, V., 2016b. Fast global registration. In *ECCV*. (cited on page 10)

Appendix

A Gradients

A.1 Adaptive Robust Pooling

- Pseudo-Huber pooling:

  arg min objective:

$$f(x,c,y) = \sum_{i=1}^{n} c^2 \left(\sqrt{1 + \left(\frac{y-x_i}{c}\right)^2} - 1 \right)$$

  Intermediate steps:

$$\begin{aligned} \mathrm{D}^2_{YY} f(x,c,y) &= \sum_{i=1}^{n} \left(1 + \left(\frac{y-x_i}{c}\right)^2\right)^{-3/2} \\ \mathrm{D}^2_{XY} f(x,c,y) &= \mathbf{vec}\left\{ -\left(1 + \left(\frac{y-x_i}{c}\right)^2\right)^{-3/2} \right\}^T \\ \mathrm{D}^2_{CY} f(x,c,y) &= \sum_{i=1}^{n} \frac{(y-x_i)^3}{\frac{(y-x_i)^2}{\alpha^2}+1)^{\frac{3}{2}} c^3} \end{aligned}$$

  Gradients:

$$\begin{aligned} \mathrm{D}_X y(x,c) &= \mathbf{vec}\left\{ \frac{w_i}{\sum_{j=1}^{n} w_j} \right\}^T \\ &\quad \text{where } w_i = \left(1 + \left(\frac{y-x_i}{c}\right)^2\right)^{-3/2} \\ \mathrm{D}_{CY} y(x,c) &= \sum_{i=1}^{n} \sum_{j=1}^{n} ((1 + \tfrac{y-x_i}{c})^2)^{\frac{3}{2}} \left(\frac{(y-x_j)^3}{((\frac{y-x_j}{c})^2+1)^{\frac{3}{2}} c^3}\right) \end{aligned} \tag{1}$$

- Huber pooling:

- Welsch pooling:

  arg min objective:

$$f(x,c,y) = \sum_{i=1}^{n}(1-\exp(-\frac{(y-x_i)^2}{2c^2}))$$

  Intermediate steps:

$$\mathrm{D}^2_{YY}f(x,c,y) = \sum_{i=1}^{n}\frac{c^2-(y-x_i)^2}{c^4}\exp\left(-\frac{(y-x_i)^2}{2c^2}\right)$$

$$\mathrm{D}^2_{XY}f(x,c,y) = \mathbf{vec}\left\{\frac{(y-x_i)^2-c^2}{c^4}\exp\left(-\frac{(y-x_i)^2}{2c^2}\right)\right\}^T$$

$$\mathrm{D}^2_{CY}f(x,c,y) = -\sum_{i=1}^{n}\frac{\left(2(y-x_i)c^2-(y-x_i)^3\right)exp(-\frac{(y-x_i)^2}{2c^2})}{c^5}$$

  Gradients:

$$\mathrm{D}_C y(x,c) = \mathbf{vec}\left\{\frac{w_i}{\sum_{j=1}^{n}w_j}\right\}^T$$

$$\mathrm{D}_X y(x,c) = \frac{v}{\sum_{i=1}^{n}w_i}$$

$$\text{where } w_i = \frac{c^2-(y-x_i)^2}{c^4}\exp\left(-\frac{(y-x_i)^2}{2c^2}\right)$$

$$v = \sum_{i=1}^{n}\frac{\left(2(y-x_i)c^2-(y-x_i)^3\right)exp(-\frac{(y-x_i)^2}{2c^2})}{c^5} \tag{2}$$

- General and adaptive pooling:

arg min objective:

$$\phi(x,\alpha,c,y) = \sum_{i=1}^{n}\left(\frac{|\alpha-2|}{\alpha}\left(\left(\frac{\left(\frac{y-x_i}{c}\right)^2}{|\alpha-2|}+1\right)^{\frac{\alpha}{2}}-1\right)\right)$$

Intermediate steps:

$$\begin{aligned}
\mathrm{D}^2_{YY} f(x,\alpha,c,y) &= \sum_{i=1}^{n} w_i \\
\mathrm{D}^2_{XY} f(x,\alpha,c,y) &= \mathbf{vec}\left\{w_i\right\} \\
\text{where } w_i &= \left(\frac{|\alpha-2|\left((\alpha-1)\,z_i^2+|\alpha-2|\,c^2\right)\left(\frac{z_i^2}{|\alpha-2|c^2}+1\right)^{\frac{\alpha}{2}}}{\left(z_i^2+|\alpha-2|\,c^2\right)^2}\right) \\
\mathrm{D}^2_{CY} f(x,\alpha,c,y) &= \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\left(\frac{z_i}{\alpha}\right)^2+1\right)^{\frac{3}{2}}\left(\frac{(z_j)^3}{\left(\left(\frac{z_j}{\alpha}\right)^2+1\right)^{\frac{3}{2}}\alpha^3}\right) \\
\mathrm{D}^2_{\alpha Y} f(x,\alpha,c,y) &= \sum_{i=1}^{n}\frac{z_i\left(\frac{z_i^2}{c^2|\alpha-2|}+1\right)^{\frac{\alpha}{2}-1}\left(\frac{\left(\frac{z_i^2}{c^2|\alpha-2|}+1\right)}{2}-t_i\right)}{c^2} \\
&\quad \text{where } z_i = y - x_i \\
&\quad \text{and } t_i = \frac{z_i^2\left(\frac{\alpha}{2}-1\right)}{c^2\left(\frac{z_i^2}{c^2|\alpha-2|}+1\right)|\alpha-2|(\alpha-2)}
\end{aligned}$$

Gradients:

$$\begin{aligned}
\mathrm{D}_X y(x,\alpha,c) &= -\frac{\mathrm{D}^2_{XY} f(x,\alpha,c,y)}{\mathrm{D}^2_{YY} f(x,\alpha,c,y)} \\
\mathrm{D}_C y(x,\alpha,c) &= -\frac{\mathrm{D}^2_{CY} f(x,\alpha,c,y)}{\mathrm{D}^2_{YY} f(x,\alpha,c,y)} \\
\mathrm{D}_\alpha y(x,\alpha,c) &= -\frac{\mathrm{D}^2_{\alpha Y} f(x,\alpha,c,y)}{\mathrm{D}^2_{YY} f(x,\alpha,c,y)}
\end{aligned} \tag{3}$$

A.2 Adaptive Feature Projections

- L_1 sphere projection:

$$
\begin{aligned}
&\text{arg min objective:} \\
&\qquad f(x,r,y) &&= \tfrac{1}{2}\|u-x\|_2^2 \\
&\text{Constraint:} \\
&\qquad h(r,y) &&= \|u\|_1 - r \\
&\qquad &&= \textstyle\sum_{i=1}^{n} |u_i| - r \\
&\text{Intermediate steps:} \\
&\qquad \mathrm{D}_Y h(r,y) &&= \mathbf{vec}\left\{\mathbf{sign}\left(y_i\right)\right\}^T \\
&\qquad \mathrm{D}^2_{YY} h(r,y) &&= 0_{n\times n} \\
&\qquad \lambda &&= \mathbf{sign}\left(y_i\right)\left(y_i - x_i\right)\ \forall i \\
&\text{Gradients:} \\
&\qquad \mathrm{D}_X y(x,r) &&= I - \tfrac{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y) \mathrm{D}_Y h(r,y)^T} \\
&\qquad \mathrm{D}_R y(x,r) &&= \tfrac{\mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}
\end{aligned}
\tag{4}
$$

- L_2 sphere projection:

$$
\begin{aligned}
&\text{arg min objective:} \\
&\qquad f(x,r,y) &&= \tfrac{1}{2}\|u-x\|_2^2 \\
&\text{Constraint:} \\
&\qquad h(r,u) &&= \|u\|_2 - r \\
&\qquad &&= \sqrt{\textstyle\sum_{i=1}^{n} u_i^2} - r \\
&\text{Intermediate steps:} \\
&\qquad \mathrm{D}_Y h(y,r) &&= y \\
&\qquad \mathrm{D}^2_{YY} h(r,y) &&= I - yy^T \\
&\qquad \lambda &&= 1 - \|x\|_2 \\
&\text{Gradients:} \\
&\qquad \mathrm{D}_X y(x,r) &&= \frac{1}{\|x\|_2}\left(I - yy^T\right) \\
&\qquad \mathrm{D}_R y(x,r) &&= \tfrac{y}{\|y\|_2} \\
&\qquad &&= \tfrac{\frac{r}{\|x\|_2} x}{\|\frac{r}{\|x\|_2} x\|_2} \\
&\qquad &&= \tfrac{x}{\|x\|_2}
\end{aligned}
\tag{5}
$$

- L_∞ sphere projection:

  arg min objective:

$$f(x,r,y) = \frac{1}{2}\|u-x\|_2^2$$

  Constraint:

$$\begin{aligned} h(r,y) &= \|u\|_\infty - r \\ &= \max_i\{|u_i|\} - r \end{aligned}$$

  Intermediate steps:

$$\begin{aligned} \mathrm{D}_Y h(r,y) &= \mathbf{vec}\left\{[\![i \in I^\star]\!]\,\mathbf{sign}\,(y_i)\right\}^T \\ &\quad \text{where } I^\star = \{i \mid |y_i| \geq |y_j| \forall j\} \\ \mathrm{D}^2_{YY} h(r,y) &= 0_{n\times n} \\ \lambda &= [\![i \in I^\star]\!]\,\mathbf{sign}\,(y_i)\,(y_i - x_i)\;\forall i \in I^\star \end{aligned}$$

  Gradients:

$$\begin{aligned} \mathrm{D}_X y(x,r) &= I - \frac{\mathrm{D}_Y h(r,y)^T \mathrm{D}_Y h(r,y)}{\mathrm{D}_Y h(r,y)\mathrm{D}_Y h(r,y)^T} \\ \mathrm{D}_R y(x,r) &= \mathbf{vec}\left\{z_i\right\} \\ &\quad \text{where } z_i = \begin{cases} 0 & \text{if } \mathrm{D}_Y h(r,y)_i = 0 \\ \mathbf{sign}\,(\mathrm{D}_Y h(r,y)_i) & \text{otherwise} \end{cases} \end{aligned} \tag{6}$$

B Experiments

B.1 Point Cloud Classification

O	Top-1 Accuracy %			Mean Average Precision×100		
%	PH(o)	PH(r)	PH(a)	PH(o)	PH(r)	PH(a)
0	84.7	**85.7**	**85.7**	95.0	95.6	**95.7**
10	85.6	86.2	**86.4**	94.6	95.1	**95.7**
20	84.8	85.5	**85.9**	95.0	95.0	**95.6**
50	83.1	83.6	**84.5**	93.5	94.9	**95.2**
90	63.4	63.1	**65.7**	78.7	76.8	**81.0**

Table 1: The impact of adaptive pseudo-Huber pooling layer on ModelNet40 and PointNet. Outliers (O) are carried in training and testing. *robust_type* represents one of PH (pseudo-Huber), H (Huber), and W (Welsch). *robust_type*(o) and *robust_type*(r) are the results reported and reproduced from [Gould et al., 2020]. *robust_type*(a) is the result from our adaptive pooling, where c is initialized as 1 and learned via backpropagation. The bold font represents the best results in rows. All experiments were conducted on ModelNet40.

O	Top-1 Accuracy %			Mean Average Precision×100		
%	H(o)	H(r)	H(a)	H(o)	H(r)	H(a)
0	86.3	85.9	**86.8**	95.4	95.5	**95.7**
10	85.5	85.9	**87.0**	95.1	95.5	**95.8**
20	85.2	85.9	**86.9**	95.0	95.6	**95.8**
50	83.9	83.5	**84.5**	94.3	**95.4**	94.8
90	63.1	63.3	**65.4**	78.5	77.5	**77.9**

Table 2: The results for adaptive Huber pooling layer. Other settings follow Table 1.

O	Top-1 Accuracy %			Mean Average Precision×100		
%	W(o)	W(r)	W(a)	W(o)	W(r)	W(a)
0	86.1	**87.1**	85.5	95.0	**95.6**	95.0
10	**86.6**	**86.6**	85.7	94.6	**95.1**	94.5
20	86.3	**86.5**	85.5	94.8	**95.7**	94.7
50	84.3	84.4	**84.9**	94.8	**95.5**	94.9
90	65.3	65.4	**66.4**	76.6	**80.5**	79.8

Table 3: The results for adaptive Welsch pooling layer. Other settings follow Table 1.

O	Top-1 Accuracy %			Mean Average Precision×100		
%	PH(o)	PH(r)	PH(a)	PH(o)	PH(r)	PH(a)
0	84.7	85.8	86.6	95.0	**95.9**	95.0
1	84.7	85.7	**86.6**	95.1	**95.8**	95.0
10	84.6	85.5	**86.4**	94.8	**95.6**	94.7
20	82.8	83.6	**84.9**	93.4	**94.6**	94.0
30	74.2	77.5	**79.5**	89.5	90.9	**91.4**
40	59.1	66.5	**69.4**	80.2	86.0	**86.5**
50	36.0	52.4	**52.8**	60.2	72.5	**76.5**
60	16.2	30.3	**34.4**	36.3	55.9	**58.7**
70	6.33	14.5	**20.4**	19.3	38.9	**40.3**
80	4.51	7.50	**9.20**	8.91	23.1	**27.2**
90	4.06	4.80	**5.40**	5.98	11.8	**17.2**

Table 4: The impact of adaptive pseudo-Huber pooling layer on PointNet and ModelNet40. Outliers (O) are carried in testing only. The remaining settings are consistent with Table 1.

O	Top-1 Accuracy %			Mean Average Precision×100		
%	H(o)	H(r)	H(a)	H(o)	H(r)	H(a)
0	86.3	**86.7**	**86.7**	95.4	95.6	**95.7**
1	86.4	86.5	**86.8**	95.3	95.6	**95.7**
10	85.3	86.1	**86.4**	94.4	**95.4**	**95.4**
20	81.1	**84.4**	83.8	92.7	**94.6**	**94.6**
30	72.2	**78.0**	**78.0**	85.1	**91.8**	91.5
40	55.4	66.3	**68.9**	72.7	**87.1**	85.4
50	34.6	52.9	**58.8**	60.1	72.0	**75.5**
60	18.1	34.4	**41.6**	38.5	60.1	**64.4**
70	7.95	15.7	**21.2**	18.4	45.8	**49.0**
80	5.64	**8.10**	6.70	8.98	25.8	**35.9**
90	4.30	**4.80**	**4.80**	5.80	13.9	**18.9**

Table 5: The results for adaptive Huber pooling layer. Other settings follow Table 4.

O	Top-1 Accuracy %			Mean Average Precision×100		
%	W(o)	W(r)	W(a)	W(o)	W(r)	W(a)
0	86.1	**86.6**	85.9	95.0	95.0	**95.5**
1	86.2	**86.8**	85.8	95.1	95.0	**95.5**
10	86.0	**86.6**	**85.6**	**94.9**	**94.9**	94.8
20	84.7	**84.9**	83.9	94.2	**94.4**	93.9
30	77.6	79.9	**81.8**	90.9	**92.6**	92.5
40	63.1	71.6	**74.1**	83.2	**88.0**	**88.0**
50	44.1	58.6	**62.5**	66.4	**80.9**	**80.9**
60	27.1	41.2	**48.0**	42.7	65.9	**66.7**
70	14.1	23.9	**28.7**	25.7	**49.1**	48.8
80	8.88	9.80	**13.6**	14.9	30.5	**35.8**
90	5.68	6.00	**7.10**	8.37	14.4	**17.1**

Table 6: The results for adaptive Welsch pooling layer. Other settings follow Table 4.

B.2 Image Classification

CIFAR10 Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
ResNet-18	92.95	-	90.27	-
ResNet-18-L_1 Sphere	**93.09**	92.91	91.53	**92.88**
ResNet-18-L_2 Sphere	93.24	**93.32**	92.87	**94.15**
ResNet-18-L_1 Ball	**93.02**	92.64	88.97	**93.07**
ResNet-18-L_2 Ball	93.41	**93.48**	90.93	**94.34**

Table 7: The impact of adaptive projection layers on ResNet-18 and CIFAR10. We report Top-1 accuracy (Top-1 Acc. %) and mean average precision (mAP ×100). All models are trained from scratch. Top-1(f) and mAP(f) is the result from baseline, where the radius are set as constant. Top-1(a) and mAP(a) is the result of adaptive feature projections (except for ResNet-18 with no projection layer). We indicate better result in non-adaptive and adaptive projections in bold font (row comparison). We indicate best result overall with red color.

Imagewoof Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
ResNet-18	62.23	-	56.57	-
ResNet-18-L_1 Sphere	79.84	**80.99**	74.08	**78.55**
ResNet-18-L_2 Sphere	79.61	**81.95**	76.78	**80.51**
ResNet-18-L_1 Ball	**80.35**	79.23	72.62	**77.75**
ResNet-18-L_2 Ball	80.35	**81.34**	72.86	**80.35**

Table 8: The impact of adaptive projection layers on ResNet-18 and Imagewoof. Other settings are consistent with Table 7.

CIFAR10 Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
GoogLeNet	92.27	-	90.39	-
GoogLeNet -L_1 Sphere	91.26	**91.88**	91.48	**93.18**
GoogLeNet -L_2 Sphere	91.20	**92.64**	92.15	**94.07**
GoogLeNet -L_1 Ball	91.04	**92.24**	88.87	**93.13**
GoogLeNet -L_2 Ball	91.29	**92.76**	91.94	**94.60**

Table 9: The impact of adaptive projection layers on GoogLeNet and CIFAR10. Other settings are consistent with Table 7.

Imagewoof Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
GoogLeNet	**79.28**	-	71.36	-
GoogLeNet -L_1 Sphere	70.93	**76.18**	64.72	**74.72**
GoogLeNet -L_2 Sphere	74.14	**78.88**	71.76	**79.81**
GoogLeNet -L_1 Ball	73.15	**75.26**	67.00	**75.15**
GoogLeNet -L_2 Ball	73.28	**79.10**	68.82	**78.75**

Table 10: The impact of adaptive projection layers on GoogLeNet and Imagewoof. Other settings are consistent with Table 7.

CIFAR10 Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
DenseNet-121	94.08	-	93.88	-
DenseNet-121 -L_1 Sphere	94.99	**95.25**	95.31	**95.94**
DenseNet-121 -L_2 Sphere	93.73	**95.18**	94.12	**96.52**
DenseNet-121 -L_1 Ball	94.29	**95.02**	94.45	**95.71**
DenseNet-121 -L_2 Ball	94.69	**94.92**	95.30	**95.73**

Table 11: The impact of adaptive projection layers on DenseNet-121 and CIFAR10. Other settings are consistent with Table 7.

Imagewoof Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
DenseNet-121	76.48	-	73.30	-
DenseNet-121 -L_1Sphere	82.13	**84.53**	74.99	**81.82**
DenseNet-121 -L_2Sphere	65.23	**85.06**	61.29	**84.28**
DenseNet-121 -L_1Ball	55.03	**84.40**	51.92	**81.90**
DenseNet-121 -L_2Ball	57.72	**80.73**	53.57	**79.28**

Table 12: The impact of adaptive projection layers on DenseNet-121 and Imagewoof. Other settings are consistent with Table 7.

CIFAR10 Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
AlexNet	89.32	-	77.77	-
AlexNet -L_1 Sphere	**89.52**	89.40	**90.93**	90.61
AlexNet -L_2 Sphere	**89.84**	89.72	**90.16**	89.67
AlexNet -L_1 Ball	89.50	**89.97**	90.82	**91.37**
AlexNet -L_2 Ball	89.61	**89.92**	**90.40**	90.23

Table 13: The impact of adaptive projection layers on AlexNet and CIFAR10. Other settings are consistent with Table 7.

Imagewoof Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
AlexNet	75.18	-	65.65	-
AlexNet -L_1 Sphere	**76.74**	76.00	**78.59**	78.57
AlexNet -L_2 Sphere	76.46	**76.99**	**79.01**	78.86
AlexNet -L_1 Ball	76.02	**76.20**	78.70	**78.76**
AlexNet -L_2 Ball	**76.38**	75.97	**77.45**	77.30

Table 14: The impact of adaptive projection layers on AlexNet and Imagewoof. Other settings are consistent with Table 7.

CIFAR10 Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
VGG-11	91.66	-	81.10	-
VGG-11 -L_1 Sphere	**90.84**	90.49	**88.41**	87.18
VGG-11 -L_2 Sphere	**91.41**	91.39	**87.02**	86.54
VGG-11 -L_1 Ball	90.21	**90.52**	87.31	**87.39**
VGG-11 -L_2 Ball	**91.37**	91.21	86.46	**87.69**

Table 15: The impact of adaptive projection layers on VGG-11 and CIFAR10. Other settings are consistent with Table 7.

Imagewoof Model	Top-1(f) Acc. %	Top-1(a) Acc. %	mAP(f) ×100	mAP(a) ×100
VGG-11	81.62	-	73.11	-
VGG-11 -L_1 Sphere	**80.17**	27.82	**82.05**	23.27
VGG-11 -L_2 Sphere	**83.13**	82.36	**83.03**	80.05
VGG-11 -L_1 Ball	**77.63**	23.06	**80.67**	16.73
VGG-11 -L_2 Ball	**82.29**	82.01	**82.16**	81.95

Table 16: The impact of adaptive projection layers on VGG-11 and Imagewoof. Other settings are consistent with Table 7. The abnormal behaviours in adaptive L_1 sphere and L_1 ball may due to exploring gradients.