RECONSTRUCTING AND ANALYZING EFFECTIVE HYPERSURFACES FROM CONVOLUTIONAL NEURAL NETWORK LAYERS USING ADJOINTBACKMAP

A Dissertation

by

QING WAN

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2022

Major Subject: Computer Science and Engineering

ABSTRACT

There are several methods in the exploration of Convolutional Neural Network's (CNN's) inner workings. However, in general, finding the inverse of the function performed by CNN as a whole is an ill-posed problem. We propose an Adjoint Operator-based method to reconstruct, given an arbitrary unit in the CNN (except for the first convolutional layer), its effective hypersurface in the input space that replicates the unit's decision surface conditioned on a particular input image. We gradually study CNN's inner workings through two steps. First, we consider a CNN without any bias for the reconstruction, which reduces the difficulties in the analysis. Next, we embed input images into an enlarged space (that considers bias as a part of the input) to enable the reconstruction of CNN's processing that includes bias vectors. Both steps confirm that any reconstructed effective hypersurface would give nearly the exact output value of that CNN unit when an inner product is computed with the original input. Also, we find that CNN unit's decision is primarily conditioned on the input. Further analysis in adversarial attacks reveals that CNN's decision is very sensitive and brittle, explaining why adversarial examples can effectively deceive CNNs.

DEDICATION

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Yoonsuck Choe, for his years' guidance during my Ph.D. study at Texas A&M University. I appreciate his ideas, time, and efforts in helping me conduct research and complete this dissertation.

Second, I would like to thank my committee members, Professor James Caverlee, Professor Sing-Hoi Sze, and Professor Jianxin Zhou, for their insightful comments on my research. Besides, I would like to express my gratitude to Professor Maurice Rojas for his kindness and help in my oral prelim.

My fellow labmates and friends much more enriched my Ph.D. life at Texas A&M. I appreciate their helpful discussions and suggestions that motivate my study and research. Additionally, I would like to give special thanks to Khuong Nguyen for his encouragement and mentoring, and Siu Wun Cheung for his technical advice.

Finally, my profound gratitude goes to my family, my wife, parents, and parents-in-law for their care and patience. My confidence comes from their incentives. I deeply appreciate their love.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Yoonsuck Choe, Professor James Caverlee, Professor Sing-Hoi Sze of the Department of Computer Science and Engineering, and Professor Jianxin Zhou of the Department of Mathematics at Texas A&M University.

All work for the dissertation was completed by the student under the advisement of Professor Yoonsuck Choe of the Department of Computer Science and Engineering at Texas A&M University.

Funding Sources

The Ph.D. study was partially supported by assistantships from the Department of Computer Science and Engineering at Texas A&M University.

NOMENCLATURE

$\circledast$	Convolution
$\mathcal{X}$	A vector space
$\boldsymbol{\theta}_{\mathcal{X}}$	The origin of the vector space $\mathcal{X}$
$\times$	A cartesian product
$[\boldsymbol{x}_i; \boldsymbol{x}_b]$	A concatenation of two tensors, $\boldsymbol{x}_i$ and $\boldsymbol{x}_b$
$\boldsymbol{x}[\boldsymbol{b}_i]$	An operation that retrieves the corresponding part of $\boldsymbol{b}_i$ from $\boldsymbol{x}$ ($\boldsymbol{x}$ is a sequential concatenation of tensors $\{\boldsymbol{b}_i\}$)
$\langle \boldsymbol{x} \mid \boldsymbol{y} \rangle$	The inner product of $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{X}$
$\mathcal{X}^*$	The algebraic dual of $\mathcal{X}$, i.e., the space of all linear functionals on $\mathcal{X}$
$\langle \boldsymbol{x}, \boldsymbol{x}^* \rangle$	The value of a linear functional $\boldsymbol{x}^* \in \mathcal{X}^*$ at $\boldsymbol{x} \in \mathcal{X}$
$B(\mathcal{X}, \mathcal{Y})$	The space of all bounded linear operators from $\mathcal{X}$ to $\mathcal{Y}$
"Eq"	An equation in the main text
"eq"	An equation in the appendix
"Algorithm"	An algorithm in the main text
"Table"	A table in the main text
"table"	A table in the appendix
"Fig"	A figure in the main text
"fig"	A figure in the appendix
"RM"	An abbreviation of Reconstruction Mode

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION AND LITERATURE REVIEW

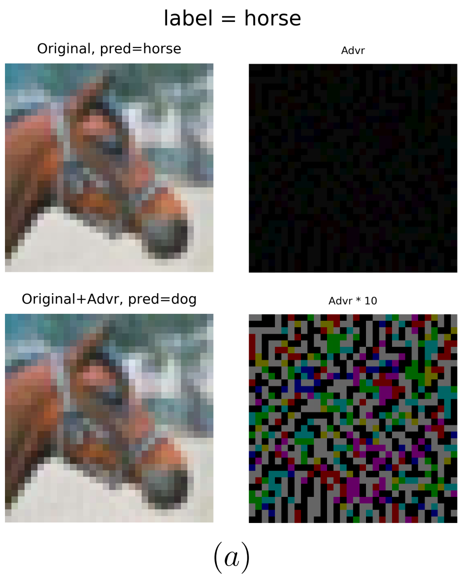
1.1 Introduction

1.1.1 Overview

Convolutional Neural Networks (CNNs) have seen great success in computer vision (CV). It is inspired by receptive fields found in biological vision [6], implemented by convolution [7], and trained by backpropagation [8]. Hardware advancements like GPUs [9] have enabled the fast training of CNNs on large datasets. Architectural evolution, such as AlexNet [10], VGG [11], ResNet [12], NASNet [13] have uplifted CNN's generalization capability, securing their dominance in the machine learning field.

Despite many successes in CV, CNN's inner workings remain largely unexplained. That mystery attracts tremendous studies probing into the black box. Early attempts found that CNNs gradually learn more complex visual features through their multi-layer convolutional architectures. Deconvolution [14] can illustrate what is represented in each layer: edge features at lower layers and object features at higher layers. CAM [15] and its successor, Grad-CAM [16] can highlight specific features in the input image that contributes most to the decision made by CNN. Especially, a guided backpropagation proposed in [16, 17] locates input features that directly influence the output, indicating which specific part the CNN used for its final decision.

However, it is well-known that CNNs are vulnerable to adversarial attacks [2, 3]. Usually, these adversarial patterns, being imperceptible to our human vision, can unexpectedly result in catastrophic classification or recognition failures. For example, Fig.1.1(a) shows an adversarial pattern that makes a prevalent ResNet20 mistakenly recognize a horse as a dog by simply adding low-level noise. Furthermore, we can fool the model to output an arbitrary label in a classification task as long as we add targeted noise to the "horse" image. Recent research [18] showed that a small physical pattern, pasted on a T-shirt (a T-shirt worn by a human with an adversarial pattern), could evade a CNN-based YOLO-v2 person detector. Nevertheless, these contaminated examples are either visually identical or easily identified to our human vision, while the well-trained CNN turns a blind eye to them.



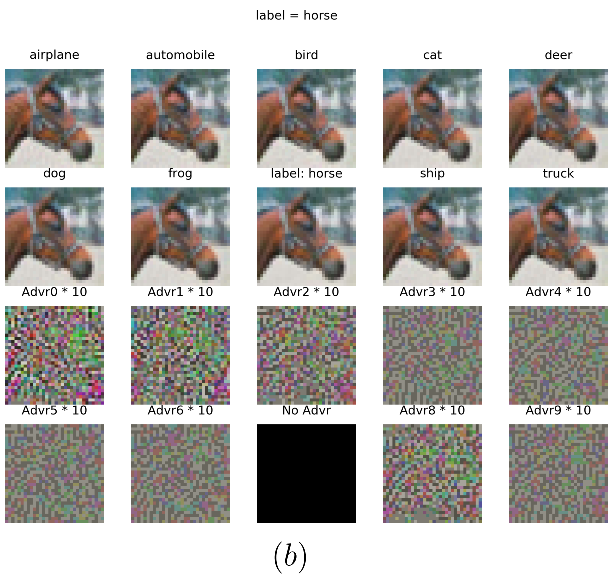


Figure 1.1: Adversarial attacks on the ResNet20 CNN model [1]. (a): A targetless attack makes the CNN mistakenly recognize an input "horse" as a "dog". The upper left corner illustrates the original "horse" image; The upper-right corner produces adversarial noise using [2]; The lower-left corner is the contaminated image using a summation of the two; The lower right corner is the noise multiplied by a factor of 10. (b): A targeted attack can fool the CNN to output any intended class as long as the injected noise pattern is appropriately generated. The lower two rows illustrate 9 noise patterns as described in [3] (scaled by $10\times$); The upper two rows illustrate the images which are the sums of the "horse" image and its corresponding adversarial noise from the lower two rows; A label in the upper two rows reports the CNN's prediction when we feed that contaminated image into ResNet20 (The image labeled "horse" does not contain any noise, i.e., "No Advr" was added).

1.1.2 Motivation

Explainable CNN is facing a non-negligible challenge from adversarial attacks. An adversarial example can fool CNN to misclassify an object and, at the same time, manipulate the basis of the CNN's prediction [19, 20], to misinterpret the object's features, despite the same visual features being shared with its original image. Recently, a study [21] found that popular interpretation algorithms do not necessarily show the correct reasoning used in CNN's predictions. These conflicts motivate us to study CNN's inner workings and explore its decision processes. Specifically, we ask three fundamental questions:

- Given a unit from a convolved feature map or a class output from the fully-connected (FC) layer inside a CNN, how is the output value factorized in the input end?
- What happens inside a CNN when an adversarial example strikes?
- Why is a CNN model vulnerable to adversarial examples?

Theoretically, these questions demand systematic modeling of CNN. The modeling is expected to correctly reflect what features or pixels the CNN recruits from an input image for its prediction and present robustness to various adversarial attacks.

1.1.3 Approach

CNN stacks layers of convolutions and nonlinear activation functions, which are cumbersome for analysis. To overcome this, we propose a novel algorithm based on Adjoint Operators (Adjoint-BackMap) that aims at precisely reconstructing effective hypersurfaces (an effective hypersurface is a function of the input that produces hyperplanes) of CNN units. Our mathematical proof and experiments have verified that:

1. Given an arbitrary unit in the CNN, we can precisely replicate its output value through a dot product between the input and an effective hyperplane generated from the reconstructed effective hypersurface.
2. Our reconstruction algorithm faithfully reproduces an arbitrary unit's activation value regardless of the input being normal or adversarial.

Based on that, we study the CNN and adversarial attacks through three steps:

1. We first consider precisely modeling the CNN without bias so that the difficulty of analysis could be appropriately reduce.

2. Furthermore, we consider slightly extending the input space to include bias as part of the input space so that we could reuse our analysis framework above.

3. Finally, we apply our theory to analyze CNN models under adversarial examples.

1.2 Related Works

1.2.1 Interpretable CNN

Understanding CNN's inner workings is an active research direction towards explainable AI. In general, explainable AI intends to find concrete features employed by CNN's prediction that shares similarities with our human eyes. Recent studies can be summarized into three tracks: inversion, perturbation, and activation map.

Inverse methods usually try to invert the CNN model. These methods invert a learned feature map in CNN back to the input space to visualize parts of the input image contributing to the feature map's output. Early attempts included Deconvolutions [14] that deconvolved a layer's feature maps to reproduce the local inputs, Guided-BP [17] that backpropagated a feature map back to input using reversed gradients, and inverse approximation [22] that used estimation to revert a high-level feature map from the input perspective.

Perturbation is another path to estimate feature importance inside a CNN. It treats a deep learning model as a black box and observes how prediction changes when input varies. Essentially, this method is similar to gradients or saliency maps [23, 24, 25, 26, 27, 28, 29, 30, 31]. A variant in this path is to maximize a class logit by iterating the CNN's input, and the final pattern returned from the iteration will be claimed as the feature visualized for the prediction [32].

Activation map was first introduced in [15, 16]. It generates a weighted activation map, a channel-wise summation of feature maps (pulled from the last layer right before global pooling) multiplied by the corresponding fully-connected (FC) layer's weights to highlight the contributing area for an explanation. Usually, this weighted activation map is smaller than the input image since poolings are usually employed inside a CNN. For example, consider an input image having a resolution of $32 \times 32 \times 3$. Its weighted activation map will generate a heatmap with a resolution of

8×8 from a CNN to be over the input image and to highlight features contributing to the CNN's decision. Usually, a resizing step ($8 \times 8 \rightarrow 32 \times 32$) is necessary for that activation map before overlaying on the input image.

Generally, these methods estimate features that contribute to CNN's decision. Therefore, weaknesses are inevitable since the estimations contain relative errors that might lead to misinterpretation. As we know, CNN is theoretically not invertible because bijection does not hold in the model. Inverting a feature map for analysis violates this principle. Also, CNN takes a high-dimensional image to activate nonlinear functions layer by layer, which implies essential features derived from perturbation might be a small part of the prediction factors. A vulnerability in activation maps comes from the necessary resizing step for the weighted activation map. That resizing implies that every 4×4 square of the input image corresponds to a unit of the activation map. However, CNN is composed of multi-layer convolutions, and its effective receptive field, a receptive region in the input image that contributes to a unit's computation [33], gradually gets enlarged as the layer goes high. The unit's effective receptive field, pulled from a high-level layer, is actually much larger than the assumed 4×4. Even the whole image area might be responsible for the unit. Therefore, applying the resized activation map to the input image may violate the effective receptive field principle.

As we mentioned, a recent study [21] found that popular interpretation algorithms do not necessarily show the correct underlying principle for CNN's prediction. Therefore, it is still too early to claim whether CNN is explainable or not.

1.2.2 Theories of CNN

CNN's internal working principle remains unclear despite many explainable trials. As we mentioned, [34, 19, 20] found that interpretation of neural network is fragile, and adversarial examples manipulate an interpretable suggestion to any desired patterns with ease, which leaves a hard hit to explainable AI because of missing comprehensive mathematical supports. However, an accurate theory for CNN can not be found overnight. Early attempts [35, 36] approached CNNs using Wavelet Theory to find correlations between CNN and filter banks for better interpretations. A recent study [37] proposed a Neural Tangent Kernel (NTK) method approximating a neural network learned by gradient descent to kernel regression. However, vulnerabilities are inevitable in these methods. Wavelet theory presumed a convolutional layer as an LTI (Linear Time-Invariant) system, which is not satisfied due to the layer's bias and the nonlinear activation function. The foundation of NTK relies on an approximation between the weight update policy and the following differential equation.

$$\begin{aligned} \boldsymbol{w}(t+1) &= \boldsymbol{w}(t) - \frac{\partial L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}(t))}{\partial \boldsymbol{w}}, \\ \frac{d\boldsymbol{w}(t)}{dt} = \lim_{\epsilon \to 0} \frac{\boldsymbol{w}(t+\epsilon) - \boldsymbol{w}(t)}{\epsilon} &= -\frac{\partial L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}(t))}{\partial \boldsymbol{w}}, \end{aligned} \tag{1.1}$$

where $\boldsymbol{w}(t)$ is the weight at time t, and $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}(t))$ is the loss function with inputs $\boldsymbol{x}$ and labels $\boldsymbol{y}$. This approximation commits relative errors and whether these errors are negligible remains questionable due to high dimensional weight matrices. Besides, the NTK regime suffers theoretical challenges [38]. These imply that the theoretical foundation of CNN is not easy to establish.

1.3 Outline of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 considers CNN without bias, modeling the CNN's convolution with adjoint operators, and introduces our reconstruction theory that gives five reconstruction modes (RMs) depending on the unit's location in the CNN. Experiments proposed in the chapter verify the theory and visualize the results. Chapter 3 upgrades the theory in Chapter 2 to take all the layers' bias values as part of the input and generalizes our reconstruction algorithm to CNNs that either use batch normalization or conventional bias. Chapter 4 presents an application of our theory to explore the fundamentals of adversarial attacks on CNN models. Chapter 5 summarizes all findings in this dissertation and discusses potential research for future exploration.

2. MODELING CNN WITHOUT ANY BIAS

2.1 Overview

This chapter aims to model convolution – the architectural feature that distinguishes CNN from other deep neural networks. Convolution is usually implemented with a receptive-field-sized kernel on an in-channel (in-ch) feature map in a CNN layer. Conventionally, the kernel was modeled as a filter [35, 36]. As we know, besides the limitation mentioned in Section 1.1, CNN stands out from image classification tasks which essentially involve the geometrical separation – an application topic evolved from the Hahn-Banach theorem [39] in Dual Space. Therefore, we innovatively consider CNN's convolution from an algebraic dual perspective [5]. Then, we convert our derivations to an algorithm.

2.2 Theory

We will model the convolution of CNN theoretically first.

We define an element-wise inner product on an input image space $\mathcal{X}$ (norm induced) as

$$\langle \mathbf{x} \mid \mathbf{x}^{'} \rangle = \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{C} x_{i,j,k} x^{'}_{i,j,k}, \tag{2.1}$$

where $\mathbf{x}$ or $\mathbf{x}^{'} \in \mathcal{X}$ is an image having height H, width W, and channels C. A CNN takes an image $\mathbf{x}_0 \in \mathcal{X}$ and propagates it through the convolutional/pooling layers to produce an in-ch feature map, to be subsequently convolved with a kernel $\mathbf{w}_{r_1 \times r_2}$, with a receptive field of size $r_1 \times r_2$ (denoted by $\mathcal{Y}$). We use $\mathbf{F}(\cdot)$ to refer to the forward propagation that computes the feature map and assume that the CNN does not have any bias, implying $\mathbf{F}(\theta_{\mathcal{X}}) = \theta_{\mathcal{Y}}$. Also, we assume that the CNN is activated with either ReLU [40] or Leaky ReLU [41]. Then, we model convolution using the dual form [42, 43],

$$c = \mathbf{F}(\mathbf{x}_0) \circledast \mathbf{w}_{r_1 \times r_2} = \langle \mathbf{F}(\mathbf{x}_0), \mathbf{w}_{r_1 \times r_2} \rangle, \tag{2.2}$$

where $c \in \mathbb{R}$ denotes a unit's value in the out-channel (out-ch) feature map; $\mathbf{w}_{r_1 \times r_2}$ is a hyperplane on $\mathcal{Y}$ and $\mathbf{w}_{r_1 \times r_2} \in \mathcal{Y}^*$. The proof (Section 2.4) turns Eq.(2.2) to the equation below.

$$c = \mathbf{F}(\mathbf{x}_0) \circledast \mathbf{w}_{r_1 \times r_2} = \langle \mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))\mathbf{x}_0, \mathbf{w}_{r_1 \times r_2} \rangle, \tag{2.3}$$

where $\mathbf{z}(\mathbf{x}_0) = k\mathbf{x}_0$ establishes the second "=" with $k \in \mathbb{R}^+$; $\mathbf{J_F} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ is a Jacobian operator that maps $\mathcal{X} \to B(\mathcal{X}, \mathcal{Y})$; and $\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0)) \in B(\mathcal{X}, \mathcal{Y})$.

Considering the Hilbert space $\mathcal{X}$, we use Riesz Representation theorem [44] and an Adjoint operator of the Jacobian, $\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))$, to simplify Eq.(2.3),

$$c = \langle \mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))\mathbf{x}_0, \mathbf{w}_{r_1 \times r_2} \rangle = \langle \mathbf{x}_0 \mid \mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{r_1 \times r_2} \rangle = \langle \mathbf{x}_0 \mid (\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0)))^T \mathbf{w}_{r_1 \times r_2} \rangle, \tag{2.4}$$

where $\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0)) \in B(\mathcal{Y}^*, \mathcal{X}^*)$. As Riesz Representation suggests that $\mathcal{X}^*$ is $\mathcal{X}$ itself, the adjoint operator will map a convolutional kernel in any layer of a CNN (except for the first convolutional layer) back to the input image space, which serves as an effective hyperplane, packing all decision hyperplanes from the input, all the way forward to the specific unit in the layer's out-ch feature map. We summarize the computational procedures in Fig.2.1.

We mention two things:

1. Eq.(2.3) is not linear since $\exists \mathbf{x}, \mathbf{y} \in \mathcal{X}$, such that, $\mathbf{J_F}(\mathbf{z}(\alpha\mathbf{x}+\beta\mathbf{y}))(\alpha\mathbf{x}+\beta\mathbf{y}) \neq \alpha\mathbf{J_F}(\mathbf{z}(\mathbf{x}))\mathbf{x} + \beta\mathbf{J_F}(\mathbf{z}(\mathbf{y}))\mathbf{y}$ for scalars α, β;

2. We will view this $\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{r_1 \times r_2}$ in the dual space instead of in the input image space if the Riesz Representation is not applied. In other words, Riesz Representation frees us from visualizing two distinct spaces.

2.3 Algorithm

2.3.1 Layers considered for analysis

Generally, we deploy our AdjointBackMap on two kinds of layers in a CNN (Fig.2.2):

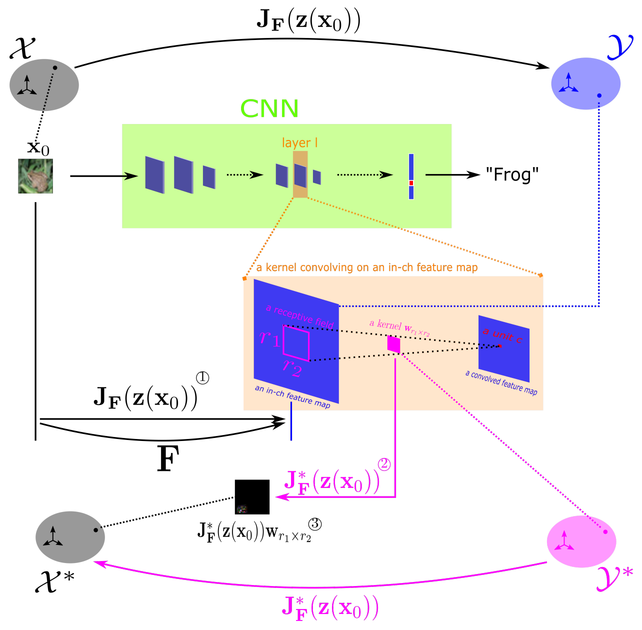

$\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))$
$\mathcal{X}$
$\mathcal{Y}$
CNN
$\mathbf{x}_0$
layer l
"Frog"
a kernel convolving on an in-ch feature map
a receptive field
r_1
r_2
a kernel $\mathbf{w}_{r_1\times r_2}$
a unit c
an in-ch feature map
a convolved feature map
$\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))$ ①
$\mathbf{F}$
$\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))$ ②
$\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{r_1\times r_2}$ ③
$\mathcal{X}^*$
$\mathcal{Y}^*$
$\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))$

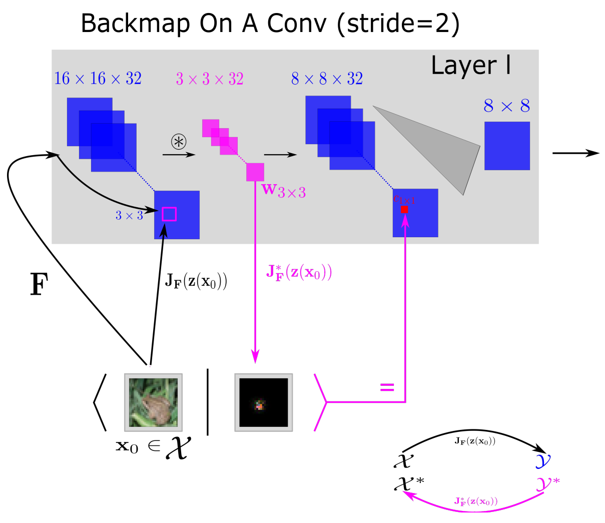
Figure 2.1: Computational procedures for applying AdjointBackMap to a unit. Given a CNN model (Green) and an input image $\mathbf{x}_0$ ("Frog"), an effective hyperplane shows how a unit c (Red) in an intermediate layer is activated from the perspective of the input space $\mathcal{X}$. The unit c's value comes from a kernel $\mathbf{w}_{r_1\times r_2}$ convolving on its in-channel (in-ch) feature map. $\mathbf{F}$ refers to the forward propagation that computes the feature map. Then, the effective hyperplane can be computed through the following procedures: ① Compute the jacobian matrix $\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))$ in Eq.(2.3); ② Transpose the matrix according to Eq.(2.4) to get its adjoint matrix $\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))$; ③ Compute the effective hyperplane through $\mathbf{J}^*_\mathbf{F}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{r_1\times r_2}$. Since we consider $\mathbf{w}_{r_1\times r_2}$ as an element in the dual of the in-ch feature map, i.e., $\mathcal{Y}^*$, the kernel is projected to the dual of the input space, i.e., $\mathcal{X}^*$, which forms an effective hyperplane that determines the unit's activity. A dotted link connects an element with the space it belongs to. Different colors distinguish different spaces.

1. We map a kernel from any convolutional layer (except for the first layer, in which case the kernel is already in $\mathcal{X}^*$) back to the input space and visualize it as an effective hyperplane that accumulates all decision hyperplanes on the forward path from the input to the activation value on the out-ch feature map at that layer (Fig.2.2(a));

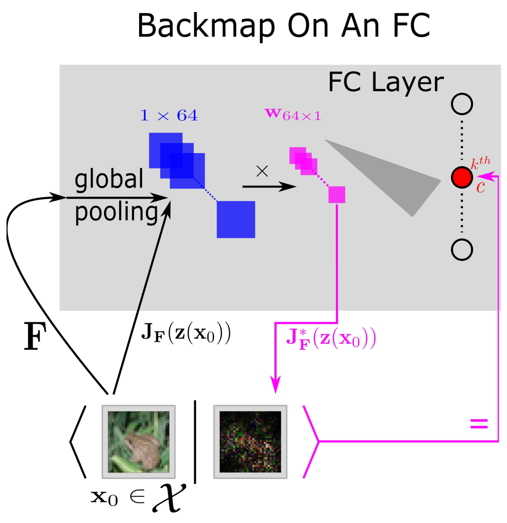
2. We map weight vectors in the FC layer back to the input image space and visualize a reconstructed hyperplane that directly determines the CNN's prediction (Fig.2.2(b)).

2.3.2 Premise

AdjointBackMap requires two necessary conditions to function as intended:



(a) Conv


Backmap On An FC
FC Layer
1×64
$\mathbf{w}_{64\times1}$
global pooling
k^{th}
c
$\mathbf{F}$
$\mathbf{J_F}(\mathbf{z}(\mathbf{x}_0))$
$\mathbf{J^*_F}(\mathbf{z}(\mathbf{x}_0))$
$\mathbf{x}_0 \in \mathcal{X}$
=

(b) FC

Figure 2.2: Principles of AdjointBackMap over a CNN. We deploy our theory on two kinds of layers in a CNN: (a) Conv layers, and (b) FC layers. Elements from the same normed space are colored identically. Specific hyperparameters such as feature map size are used for easier tracking. $\mathbf{F}$ denotes the path from the input image to an in-channel (in-ch) 3×3 receptive field (or the global pooling [4] layer) for convolving with the layer's kernel $\mathbf{w}_{3\times3}$ (or multiplying with the layer's weights $\mathbf{w}_{64\times1}$). Its adjoint operator, $\mathbf{J}^*_{\mathbf{F}}(\mathbf{z}(\mathbf{x}_0))$, maps the kernel $\mathbf{w}_{3\times3}$ (or $\mathbf{w}_{64\times1}$) back to the image space $\mathcal{X}$. The "$\langle \cdot \mid \cdot \rangle$" notation is the dot-product defined in Eq.(2.1). The symbol "$=$" means that the dot-product between the input image ($\mathbf{x}_0$) and the reconstructed effective hyperplane ($\mathbf{J}^*_{\mathbf{F}}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{3\times3}$) equals the unit's value (or the predicted class value using $\mathbf{J}^*_{\mathbf{F}}(\mathbf{z}(\mathbf{x}_0))\mathbf{w}_{64\times1}$) of a convolved feature map (or the FC layer) computed from the CNN.

1. CNN should not have any bias. Otherwise, Eq.(2.3) cannot be established appropriately[1];

2. The CNN should be activated with ReLU/Leaky ReLU/other functions whose derivatives are piecewise constants and satisfy Eq.(2.12) in Section 2.4. Therefore, care should be taken when deploying our method on a CNN using activation functions whose derivative is not piecewise constant (like tanh).

These necessary conditions ensure that the effective hypersurface reconstructed by AdjointBackMap can reproduce a CNN unit's activation, given an arbitrary input image. That is, if we dot-product (Eq.(2.1)) an effective hyperplane to the input image directly, the returned value will precisely

[1]Note that this is not a hard requirement. This condition makes our derivation and results more understandable. See Appendix of [5] for details.

match the unit's activation value obtained from the convolved feature map or the activation value from the FC layer. This is a crucial point that distinguishes our model from other methods. We also discuss two examples (Table 4.1 and 4.2), indicating that numerical precision is essential for analyzing CNNs.

2.3.3 Five Reconstruction Modes (RMs)

In practice, AdjointBackMap provides five reconstruction modes (RMs) that depend on which unit in what layer is being considered for analysis. Four of them act on a convolutional layer and one on the FC layer. We name the five RMs as $RM4$ to $RM0$. In the view of the input end, $RM4$ and $RM3$ parse a single kernel's behavior; $RM2$ and $RM1$ focus on a group of in-ch kernels' activities; $RM0$ analyzes final class output. Fig.2.3 ($RM4$ to $RM1$) and Fig.2.2 ($RM0$) illustrate the basic concepts. Conv layers have four RMs due to two factors in CNN's convolution (Fig.2.3):

1. With or without global pooling (g_p): Fig.2.3(b)&(d) [$RM3$ and $RM1$] vs. Fig.2.3(a)&(c) [$RM4$ and $RM2$], respectively;

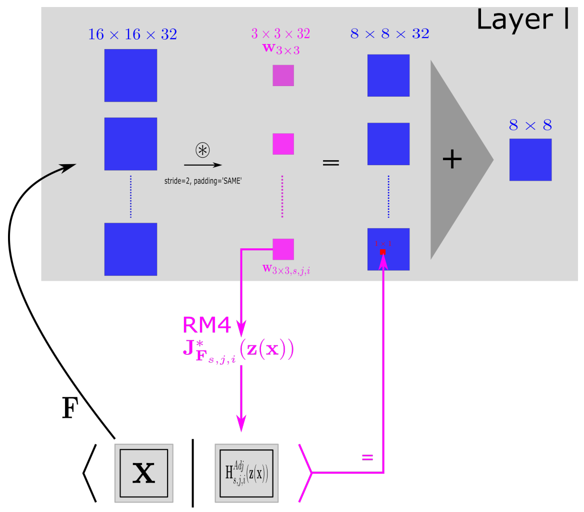
2. With or without in-ch merge during convolution: Fig.2.3(c)&(d) [$RM2$ and $RM1$] vs. Fig.2.3(a)&(b) [$RM4$ and $RM3$], respectively.

We explain using a concrete example with specific hyperparameters such as kernel size, number of channels, etc., to make it easier to track. Suppose a CNN takes a $32 \times 32 \times 3$ (height $\times$ width $\times$ channels) RGB image $\mathbf{x}$. Its third convolutional layer has in-ch feature maps of shape $16 \times 16 \times 32$. That layer has convolutional kernels of $3 \times 3 \times 32 \times 64$ (height $\times$ width $\times$ in-channels $\times$ out-channels). The convolutional stride is 2 with padding 'same' [45].

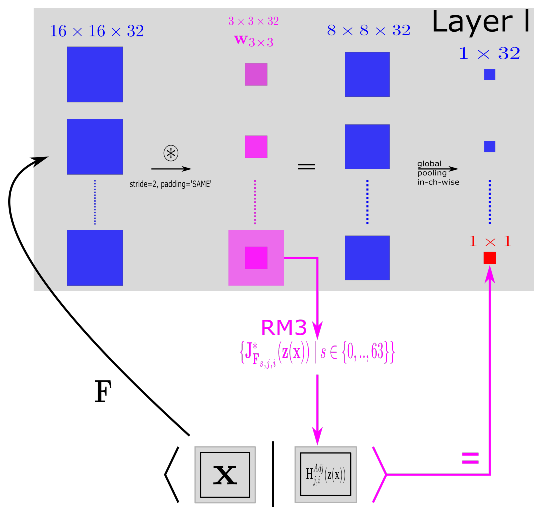
2.3.3.1 $RM4$ (Fig.2.3(a))

Convolutional kernels are mapped separately along in-channels. AdjointBackMap works on kernels, one at a time (32×64 kernels). Also, each stride move is backward mapped independently. An effective hypersurface, $\mathbf{H}^{Adj}_{s,j,i} : \mathcal{X} \to \mathcal{X}$, reconstructed through $RM4$ are (from Eq.(2.4)),

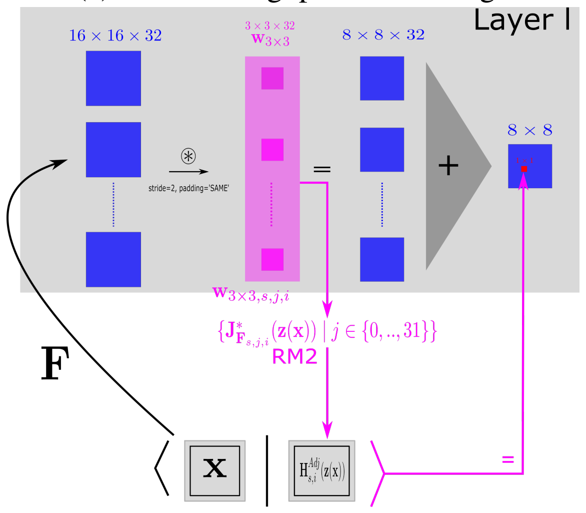
$$\mathbf{H}^{Adj}_{s,j,i}(\mathbf{z}(\mathbf{x})) = \mathbf{J}^{*}_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x}))\mathbf{w}_{3\times3,s,j,i}, \tag{2.5}$$



(a) $RM4$: w/o g_p or in-ch merge

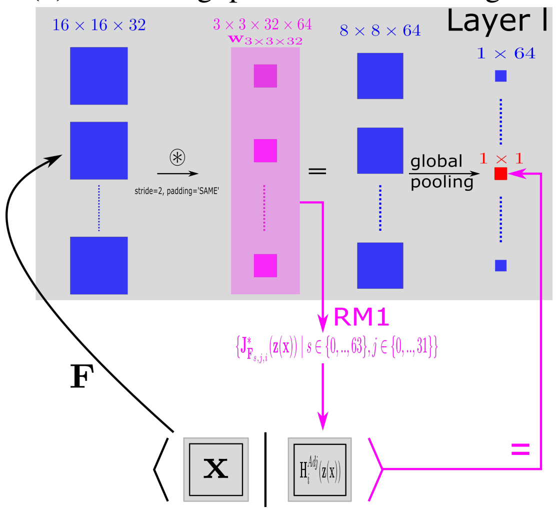


(b) $RM3$: w g_p and w/o in-ch merge



Layer l
$16 \times 16 \times 32$
$3 \times 3 \times 32$
$\mathbf{w}_{3\times3}$
$8 \times 8 \times 32$
8×8
stride=2, padding='SAME'
$\mathbf{w}_{3\times3,s,j,i}$
$\{\mathbf{J}^*_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x})) \mid j \in \{0,..,31\}\}$
RM2
F
$\mathbf{x}$
$\mathbf{H}^{Adj}_{s,i}(\mathbf{z}(\mathbf{x}))$

(c) $RM2$: w/o g_p and w in-ch merge



Layer l
$16 \times 16 \times 32$
$3 \times 3 \times 32 \times 64$
$\mathbf{w}_{3\times3\times32}$
$8 \times 8 \times 64$
1×64
global pooling
1×1
stride=2, padding='SAME'
RM1
$\{\mathbf{J}^*_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x})) \mid s \in \{0,..,63\}, j \in \{0,..,31\}\}$
F
$\mathbf{x}$
$\mathbf{H}^{Adj}_{i}(\mathbf{z}(\mathbf{x}))$

(d) $RM1$: w both g_p and in-ch merge

Figure 2.3: Overview of $RM4 \sim RM1$ on a concrete CNN ($RM0$ is illustrated in Fig.2.2(b)). Elements from the same space are colored identically. $\mathbf{F}$ represents the path from an input $\mathbf{x}$ to in-ch feature maps. The adjoint operator, $\mathbf{J}^*_{\mathbf{F}}(\mathbf{z}(\mathbf{x}))$, projects the corresponding conv kernel ($\mathbf{w}_{3\times3,s,j,i}$) back to the input space to reconstruct an effective hypersurface $\mathbf{H}^{Adj}$. (a) $RM4$: Analyze a unit in a convolved feature map; (b) $RM3$: Analyze a unit pooled globally from an in-ch convolved feature map; (c) $RM2$: Analyze a unit in an out-ch feature map; (d) $RM1$: Analyze a unit pooled globally from an out-ch feature map.

where $s \in \{0, 1, ..., 63\}, j \in \{0, 1, ..., 31\}, i \in \{0, 1, ..., 63\}$, $\mathbf{F}_{s,j,i}$ denotes a forward path from the input to the receptive field with its $(j, i)^{th}$ kernel and a stride move s. In this case, an input image $\mathbf{x}$ brings its effective hyperplane $\mathbf{H}^{Adj}_{s,j,i}(\mathbf{z}(\mathbf{x}))$ $(\in \mathcal{X})$ of a shape $32 \times 32 \times 3$. The number of backward mappings is $8 \times 8 \times 32 \times 64$. This is the most basic application that strictly follows our theory. We can analyze what a single kernel is actually doing, from the perspective of the input image space, on its effective receptive field [33] with a stride offset.

2.3.3.2 $RM3$ (Fig.2.3(b))

Only convolutional kernels are mapped separately along in-channels. A kernel's stride moves are summed, which results in a single backward mapping. In detail, a 3×3 kernel reconstructs its effective hypersurface at a stride move s. The total stride moves of the kernel are 8×8 that is also the number of effective hypersurfaces to be summed for the single backward mapping, i.e., 64 effective hypersurfaces sum together pixel-wise to form a single $32 \times 32 \times 3$ effective hypersurface, $\mathbf{H}^{Adj}_{j,i} : \mathcal{X} \rightarrow \mathcal{X}$. And that effective hyperplane $\mathbf{H}^{Adj}_{j,i}(\mathbf{z}(\mathbf{x}))$ highlights the input area that determines the unit pooled globally from an in-ch convolved feature map.

$$\mathbf{H}^{Adj}_{j,i}(\mathbf{z}(\mathbf{x})) = \sum_{s=0}^{8\times8-1} \mathbf{J}^{*}_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x}))\mathbf{w}_{3\times3,s,j,i} = \sum_{s=0}^{63} \mathbf{H}^{Adj}_{s,j,i}(\mathbf{z}(\mathbf{x})). \tag{2.6}$$

The last "=" also shows the relationship to $RM4$.

2.3.3.3 $RM2$ (Fig.2.3(c))

Convolutional kernels are merged in-channel-wise for mapping, and the kernels' strides are separated for projections. Thus, the number of backward mappings is $8 \times 8 \times 64$, and each one has its shape of $32 \times 32 \times 3$. An effective hypersurface, $\mathbf{H}^{Adj}_{s,i} : \mathcal{X} \rightarrow \mathcal{X}$, reconstructed from $RM2$ are,

$$\mathbf{H}^{Adj}_{s,i}(\mathbf{z}(\mathbf{x})) = \sum_{j=0}^{31} \mathbf{J}^{*}_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x}))\mathbf{w}_{3\times3,s,j,i} = \sum_{j=0}^{31} \mathbf{H}^{Adj}_{s,j,i}(\mathbf{z}(\mathbf{x})). \tag{2.7}$$

Similarly, the last "=" reveals its relationship to $RM4$. This effective hypersurface shows how a unit is activated in the out-ch feature map.

2.3.3.4 $RM1$ (Fig.2.3(d))

Convolutional kernels are merged in-channel-wise. A kernel's stride moves are summed for a projection as well. Thus the total number of effective hypersurfaces is 64 (the number of channels to be pooled), and each $\mathbf{H}_i^{Adj}(\mathbf{z}(\mathbf{x}))$ has its shape of $32 \times 32 \times 3$ that carries what the in-ch kernels jointly extract from the RGB image from the global pooling perspective.

$$\begin{aligned} \mathbf{H}_i^{Adj}(\mathbf{z}(\mathbf{x})) &= \sum_{j=0}^{31} \sum_{s=0}^{8\times 8-1} \mathbf{J}^*_{\mathbf{F}_{s,j,i}}(\mathbf{z}(\mathbf{x}))\mathbf{w}_{3\times 3,s,j,i} \\ &= \sum_{j=0}^{31} \mathbf{H}_{j,i}^{Adj}(\mathbf{z}(\mathbf{x})) = \sum_{s=0}^{63} \mathbf{H}_{s,i}^{Adj}(\mathbf{z}(\mathbf{x})). \end{aligned} \tag{2.8}$$

Relationships to $RM3$ and $RM2$ are listed in the second line of Eq.(2.8).

2.3.3.5 $RM0$ (Fig.2.2(b))

AdjointBackMap deploys on the weight vectors $\{\mathbf{w}_k\}$ where k denotes class index in the FC layer of the CNN. The output value for class k (before going through an activation) is determined by an effective hyperplane $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}))$ of a shape $32 \times 32 \times 3$, i.e.,

$$\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x})) = \mathbf{J}^*_{\mathbf{F}}(\mathbf{z}(\mathbf{x}))\mathbf{w}_k, \tag{2.9}$$

where $k \in \{0, 1, ..., (N-1)\}$, with N denoting the number of output units of the CNN (the number of classes).

2.3.4 Implementation

We use convolution to compute duality in Eq.(2.4) as they are equivalent, and duality can use hardware acceleration. Due to computationally expensive Jacobian, Eq.(2.5) $\sim$ Eq.(2.9) are optimized and summarized in Algorithm 1. Note: conv2d, unstack, stack, expanddim, and matmul are functions defined in Tensorflow [45]. Padding of conv2d is the same as training. Also, conv2d has an ‘axis’ choice to replace the transpose in Eq.(2.4).

Although an effective hyperplane acquired from AdjointBackMap is in the same space as the original input image, the scalar element values of the effective hyperplane might not lie on the same interval, $[0, 1]$, as its original image. In that case, we properly normalize the value to enable its visualization. We will explain more about this in our experiments.

Algorithm 1: AdjointBackMap from $RM4$ to $RM0$

```
Input: 1. $\mathbf{x}_d$: input ($d = H \times W \times C$); 2. $\mathbf{z}$: function for Eq.(2.3); 3. $\mathbf{T}$: pre-trained
        model; 4. $l$: layer index; 5. $s$: stride during training; 6. $mode$: $RM$ to be used (one
        of $RM0$ to $RM4$).
Output: Effective hyperplane $\mathbf{H}^{Adj}(\mathbf{z}(\mathbf{x}_d))$
Function AdjointBackMap($\mathbf{x}_d, \mathbf{z}, \mathbf{T}, l, s, L$):
    $\mathbf{z}_0 = \mathbf{z}(\mathbf{x}_d)$
    switch $mode$ do
        case '$RM0$' do                                              // 1. FC layer
            load $\mathbf{w}_{fc,c_{in} \times c_{labels}}$, $\mathbf{F}_{g\_p,c_{in}}$ from $\mathbf{T}$
            $\mathbf{J}_\mathbf{F} = \frac{\partial \mathbf{F}_{g\_p,c_{in}}}{\partial \mathbf{x}_d}$
            return matmul($\mathbf{J}_\mathbf{F}(\mathbf{z_0})$, $\mathbf{w}_{fc,c_{in} \times c_{labels}}$, axis='$c_{in}$')
        case '$RM4$' or '$RM3$' do                                   // 2. Without in-ch merge
            load $\mathbf{F}_{l,H_l \times W_l \times c_{in}}$, $\mathbf{w}_{l,r_1 \times r_2 \times c_{in} \times c_{out}}$ from $\mathbf{T}$ at $l$
            $\mathbf{J}_{\mathbf{F},d \times H_l \times W_l \times c_{in}} = \frac{\partial \mathbf{F}_{l,H_l \times W_l \times c_{in}}}{\partial \mathbf{x}_d}$
            $\mathbf{w}_u$ = unstack($\mathbf{w}_{l,r_1 \times r_2 \times c_{in} \times c_{out}}$, axis='$c_{in}$')
            $\mathbf{J}_{\mathbf{F},u}$ = unstack($\mathbf{J}_{\mathbf{F},d \times H_l \times W_l \times c_{in}}$, axis='$c_{in}$')
            Empty container $R$, $j = 0$
            while $j < c_{in}$ do
                $\mathbf{J}_\mathbf{F}$ = expanddim($\mathbf{J}_{\mathbf{F},u}[j]$, axis='$c_{in}$')
                $\mathbf{w}$ = expanddim($\mathbf{w}_u[j]$, axis='$c_{in}$')
                $R$.append(conv2d($\mathbf{J}_\mathbf{F}(\mathbf{z}_0)$, $\mathbf{w}$, stride=$s$, axis='$(H_l, W_l, c_{in}, c_{out})$'))
                $j = j + 1$
            $h_{d \times H_o \times W_o \times c_{in} \times c_{out}}$ = stack($R$, axis='$c_{in}$')
            if $mode$ is '$RM4$' then                                // 2.1. without g_p
                return $h_{d \times H_o \times W_o \times c_{in} \times c_{out}}$
            else if $mode$ is '$RM3$' then                           // 2.2. with g_p
                return sum($h_{d \times H_o \times W_o \times c_{in} \times c_{out}}$, axis='$(H_o, W_o)$')
        case '$RM2$' or '$RM1$' do                                   // 3. With in-ch merge
            load $\mathbf{F}_{l,H_l \times W_l \times c_{in}}$, $\mathbf{w}_{l,r_1 \times r_2 \times c_{in} \times c_{out}}$ from $\mathbf{T}$ at $i$
            $\mathbf{J}_\mathbf{F} = \frac{\partial \mathbf{F}_{l,H_l \times W_l \times c_{in}}}{\partial \mathbf{x}_d}$
            $h_{d \times H_o \times W_o \times c_{out}}$ = conv2d($\mathbf{J}_\mathbf{F}(\mathbf{z}_0)$, $\mathbf{w}_{l,r_1 \times r_2 \times c_{in} \times c_{out}}$, stride=$s$,
             axis='$(H_l, W_l, c_{in}, c_{out})$')
            if $mode$ = '$RM2$' then                                 // 3.1. without g_p
                return $h_{d \times H_o \times W_o \times c_{out}}$
            else if $mode$ = '$RM1$' then                            // 3.1. with g_p
                return sum($h_{d \times H_o \times W_o \times c_{out}}$, axis='$(H_o, W_o)$')
    return NULL
```

2.4 Proof of Eq.(2.3)

We prove that Eq.(2.3) holds for $\mathbf{z}(\mathbf{x}) = k\mathbf{x},$ for some $k \in \mathbb{R}^+$ when the neural network (without any bias) is activated with either ReLU or Leaky ReLU. We only prove $\mathbf{F}(\mathbf{x}) = \mathbf{J_F}(\mathbf{z}(\mathbf{x}))\mathbf{x}$ since the rest is trivial.

For any single convolutional layer l, without loss of generality, a pixel p of the feature map is activated after in-channel (in-ch) kernels $\mathbf{w}_{l,r_1\times r_2\times c_{in}}$ are convolved on their in-ch receptive field feature maps $\mathbf{x}_{l-1,r_1\times r_2\times c_{in}}$ (Eq.(2.1), Eq.(2.4)),

$$\begin{aligned} c = \sum_j \left\langle \mathbf{x}_{l-1,r_1\times r_2,j} \mid \mathbf{w}_{l,r_1\times r_2,j} \right\rangle, \\ \rho = \sigma(c), \end{aligned} \tag{2.10}$$

where j denotes in-ch index and σ is an activation function (either ReLU or Leaky ReLU). Also,

$$\sum_j \left\langle \mathbf{x}_{l-1,r_1\times r_2,j} \middle| \frac{\partial c}{\partial \mathbf{x}_{l-1,r_1\times r_2,j}} \right\rangle = \sum_j \langle \mathbf{x}_{l-1,r_1\times r_2,j} \mid \mathbf{w}_{l,r_1\times r_2,j} \rangle = c. \tag{2.11}$$

From the above, we have

$$\begin{aligned} &\sum_j \left\langle \mathbf{x}_{l-1,r_1\times r_2,j} \middle| \frac{\partial p}{\partial \mathbf{x}_{l-1,r_1\times r_2,j}} \bigg|_{\mathbf{z}(\mathbf{x}_{l-1,r_1\times r_2})} \right\rangle \\ &= \sum_j \left\langle \mathbf{x}_{l-1,r_1\times r_2,j} \middle| \frac{dp}{dc} \mathbf{w}_{l,r_1\times r_2,j} \right\rangle \\ &= \begin{cases} r, & kc < 0 \\ c, & kc \geq 0 \end{cases}, k \in \mathbb{R}^+ \\ &= \rho. \end{aligned} \tag{2.12}$$

where $r = 0$ for the ReLU, or $r = -0.2c$ for the default Leaky ReLU in TensorFlow. Hence, $\mathbf{f}_l(\mathbf{x}_{l-1}) = \mathbf{J}_{\mathbf{f}_l}(\mathbf{z}(\mathbf{x}_{l-1}))\mathbf{x}_{l-1}$ holds for any receptive field of convolutional layers ($\mathbf{f}_l$ depicts a mapping from in-ch features to its activated out-channel (out-ch) features in the l^{th} layer). Also, it holds for any avg- or max-pooling layer (Avg pooling is equivalent to convolving feature map-s with an averaging kernel, and max pooling is equivalent to convolving with a one-hot kernel.) Then, stacked layers of convolutions can be simplified as multiplications ($\times$) of Jacobian matrice, i.e.,

$$\begin{aligned} \mathbf{F}(\mathbf{x}) = \mathbf{f}_l(\mathbf{x}_{l-1}) = \mathbf{J}_{\mathbf{f}_l}(\mathbf{z}(\mathbf{x}_{l-1}))\mathbf{x}_{l-1} \\ = \mathbf{J}_{\mathbf{f}_l}(\mathbf{z}(\mathbf{x}_{l-1})) \times ... \times \mathbf{J}_{\mathbf{f}_1}(\mathbf{z}(\mathbf{x}_0)) \times \mathbf{J}_{\mathbf{f}_0}(\mathbf{z}(\mathbf{x}))\mathbf{x} \\ = \mathbf{J}_{\mathbf{F}}(\mathbf{z}(\mathbf{x}))\mathbf{x}, \end{aligned} \tag{2.13}$$

which works pointwise for any image vector $\mathbf{x}$.

We mention two things:

1. The proof also works for activation functions having piecewise constant derivatives when $k = 1$ is applied;

2. Although our theory requires no bias, it does not imply we cannot have any bias at all. In fact, we can alleviate the limitation with "equivalent bias" by incorporating the bias into CNN's input. For example, considering a CNN that takes an affine image $\mathbf{x}^{'}$ (the normalization of an image $\mathbf{x}$) as the input,

$$\mathbf{x}^{'} = k\mathbf{x} + b, k > 0, \tag{2.14}$$

and generates two output values to classify a binary outcome $\{0, 1\}$. By the above proof and Eq.(2.19), its decision boundary $\mathcal{C}$ is,

$$\begin{aligned} \mathcal{C} = \{\mathbf{x} \mid \langle k\mathbf{x} + b \mid h_{k=0}(\mathbf{x}^{'})\rangle \geq \langle k\mathbf{x} + b \mid h_{k=1}(\mathbf{x}^{'})\rangle\} \\ = \{\mathbf{x} \mid k\langle \mathbf{x} \mid h_{k=0}(\mathbf{x}^{'}) - h_{k=1}(\mathbf{x}^{'})\rangle + b \times (h_{k=0}(\mathbf{x}^{'}) - h_{k=1}(\mathbf{x}^{'})) \geq 0\}, \end{aligned} \tag{2.15}$$

where $h_{k\in\{0,1\}}$ is the effective hypersurface that determines the CNN's output value. Here $b \times (h_{k=0}(\mathbf{x}^{'}) - h_{k=1}(\mathbf{x}^{'}))$ serves as an equivalent bias for the CNN's decision boundary. This equivalent form allows bias to be incorporated into our theory.

2.5 Experiments

Below, we discuss how we conduct experiments on CNN models (without bias) using AdjointBackMap.

2.5.1 Pre-trained model

We discuss how we trained our CNN models for subsequent analysis using AdjointBackMap.

2.5.1.1 Dataset

We used CIFAR-10 [46] as our dataset. The CIFAR-10 dataset contains $50,000$ 32×32 RGB (value range $[0, 1]$) images for training and $10,000$ for testing, and there are 10 classes. All images are normalized with RGB means (0.4914, 0.4822, 0.4465) and standard deviations (0.2023, 0.1994, 0.2010) [1]. We separated the $50,000$ samples into one training set and one validation set with a ratio of $9 : 1$, i.e., $45,000$ and $5,000$, respectively. All analyses were conducted on the test set.

2.5.1.2 Data augmentation

We used data augmentation for training. An input color image goes through random flipping of left to right, random adjustment of saturation within $[0.0, 2.0]$, random adjustment of contrast within $[0.4, 1.6]$, random adjustment of brightness with 0.5, resizing to $36 \times 36 \times 3$, and then randomly cropped to $32 \times 32 \times 3$.

2.5.1.3 Model

We used two models: (1) VGG [11] with 7 activation layers (VGG7, without bias) and (2) 20-layer ResNet with fixup initialization [1] [47](Fixup-ResNet20), with weight rescaling and without bias, the same as Figure 1 (Middle) of [1]. The learnable parameters of VGG7 and Fixup-ResNet20 are listed as tables A.1 and A.2 (Appendix), respectively.

2.5.1.4 Cost and accuracy

We regularized the kernels by L_1 (factor 10^{-4}). We used cross-entropy with softmax as a cost. Accuracy was measured by a prediction index (the predicted class) being exactly matched with its label (Top-1 accuracy).

2.5.1.5 Training, validation, test

We trained and validated VGG7 and Fixup-ResNet20 with GD (gradient descent) optimizer on an RTX2080Ti GPU. The batch size was 100. We list additional details in Table 2.1. We trained with different learning rates or epochs so that both models could learn sufficiently. We trained with $45,000$ samples every epoch and validated the trained model on $5,000$ samples every two epochs, and the trained models would be saved if a higher validation accuracy was achieved. We tested the models on the CIFAR-10's test set. The last column in Table 2.1 shows the test accuracy of the two models. Although these accuracy results are modest, since the main focus of our work is on the analysis of trained CNNs, these were deemed sufficient for our purpose.

Model/LR intervals	1^{st} interval	2^{nd} interval	3^{rd} interval	Total	Test Acc
VGG7	$2 \times 10^{-4}, [0, 199]$	$10^{-4}, [200, 249]$	$5 \times 10^{-5}, [250, 300]$	301	85.6%
Fixup-ResNet20	$2 \times 10^{-3}, [0, 99]$	$10^{-3}, [100, 149]$	$5 \times 10^{-4}, [150, 200]$	201	90.3%

Table 2.1: Details of training schedule (learning rate: LR) and results for VGG7/Fixup-ResNet20. "$2 \times 10^{-4}, [0, 199]$" denotes that the learning rate (LR) 2×10^{-4} was maintained during the 0^{th} to the 199^{th} training epoch.

2.5.2 Five experiments with respect to five RMs

We first verify that $k = \frac{1}{8}$ satisfies Eq.(2.3). With that, we conducted five visualization experiments related to five RMs (we only show $RM0$ in the main text, and $RM4$ to $RM1$ are shown in Appendix of [5]).

2.5.2.1 Verification of $k = \frac{1}{8}$ (i.e., $\mathbf{z}(\mathbf{x}) = k\mathbf{x} = \frac{1}{8}\mathbf{x}$) satisfying Eq.(2.3)

Generally, we use Eq.(2.1) and (2.7) to experimentally verify Eq.(2.3) on two types of layers:

1. On every convolutional layer;

2. On the fully connected (FC) layer.

2.5.2.1.1 On every conv layer: A convolutional layer (except the first layer) is equipped with kernels, $\mathbf{w}_{r_1 \times r_2 \times c_{in} \times c_{out}}$, which convolve the in-ch feature maps $(\mathbf{F}(\mathbf{x}))_{H_l \times W_l \times c_{in}}$ for an input image $\mathbf{x}$ (shape: $32 \times 32 \times 3$). The hyperplanes returned from $RM2$ of Algorithm 1 are $\{\mathbf{H}^{Adj}_{s,i}(\mathbf{z}(\mathbf{x})) \mid s \in \{0, 1, ..., (H_l \times W_l - 1)\}, i \in \{0, 1, ..., (c_{out} - 1)\}\}$. For our trained VGG7 or Fixup-ResNet20, we verify that setting $k = \frac{1}{8}$ (so that $\mathbf{H}^{Adj}_{s,i}(\mathbf{z}(\mathbf{x})) = \mathbf{H}^{Adj}_{s,i}(\frac{\mathbf{x}}{8})$) leads to

$$c_{s,i} = \underbrace{((\mathbf{F}(\mathbf{x}))_{H_l \times W_l \times c_{in}} \circledast \mathbf{w}_{r_1 \times r_2 \times c_{in}, i})_s}_{\text{A unit's actual value from a conv layer}} = \underbrace{\langle \mathbf{x} \mid \mathbf{H}^{Adj}_{s,i}(\mathbf{z}(\mathbf{x})) \rangle}_{\text{The reconstructed value through a dot-product}} = p_{s,i}, \tag{2.16}$$

where $()_s$ denotes taking a real value from $()$ at a stride move s. Verifying $RM2$ is equivalent to verifying $RM4$, $RM3$, $RM1$.

2.5.2.1.2 On the FC layer: Suppose c_k denotes an activation value for the class index k of the FC layer, i.e., $c_k = \mathbf{F}_k(\mathbf{x})$. Algorithm 1 works on the weights of the FC layer with $RM0$ to generate a hyperplane, $\mathbf{H}^{Adj}_k(\mathbf{z}(\mathbf{x}))$, where $\mathbf{z}(\mathbf{x}) = \frac{\mathbf{x}}{8}$. According to Eq.(2.9) and (2.1), we verify that,

$$c_k = \underbrace{\mathbf{F}_k(\mathbf{x})}_{\text{A predicted class value from the FC layer unit } k} = \underbrace{\langle \mathbf{x} \mid \mathbf{H}^{Adj}_k(\mathbf{z}(\mathbf{x})) \rangle}_{\text{The reconstructed class value of unit } k \text{ through a dot-product}} = p_k. \tag{2.17}$$

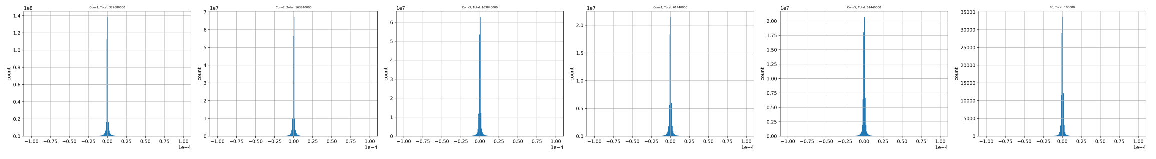
2.5.2.1.3 Experiment: We validate Eq.(2.16) and (2.17) on the CIFAR-10's $10,000$ test samples. Since errors are inevitable in decimal computation using the single-precision floating-point (32-bit float, FP32 for short) data type, we calculate the relative errors between feature map units (or 10 output units of the FC layer), $\mathbf{c}_i$, and the dot-product-based reconstructed values $\mathbf{p}_i$ (Eq.(2.16) or (2.17)) as

$$\mathbf{e}_i = \frac{\mathbf{p}_i - \mathbf{c}_i}{\mathbf{c}_{i,\neq 0}}, \tag{2.18}$$

where $i \in \{0, 1, ..., (c_{out} - 1)\}$ (or $i \in \{0, 1, ..., 9\}$) denotes the i^{th} out-ch feature map (or the i^{th} unit value of the FC layer before ReLU6); $\mathbf{c}_{i,\neq 0}$ substitutes all zeros inside $\mathbf{c}_i$ with the smallest positive number of the FP32 data type to avoid any divide by zero exception.

2.5.2.1.4 Results: We show 6 histograms of relative errors of VGG7 in Fig.2.4 (5 conv layers + 1 FC layer = 6), and 19 histograms of relative errors in Fixup-ResNet20 in Fig.2.5 (18 conv layers + 1 FC layer = 19). The size of each layer (the out-ch feature maps) is shown in the 3^{rd} column of table A.1 (VGG7) and the 4^{th} column of table A.2 (Fixup-ResNet20), Appendix. We collect a set of relative error values from each layer in the two models (a total of 6 + 19 layers), then compute the histograms from these sets. In sum, the histograms verify that $\mathbf{H}_{s,i}^{Adj}(\mathbf{z}(\mathbf{x}))$ and $\mathbf{H}_{k}^{Adj}(\mathbf{z}(\mathbf{x}))$ achieve Eq.(2.3) with over 99.99% of the relative errors being ≤ 0.01 in both convolutional layers and the FC layer of both models (VGG7 and Fixup-Resnet20). Therefore the results confirm that AdjointBackMap successfully reconstructs effective hypersurfaces that precisely determine either the feature map or the FC output.

Thus we conclude that our model satisfies Eq.(2.3) when $\mathbf{z}(\mathbf{x}) = \frac{\mathbf{x}}{8}$.

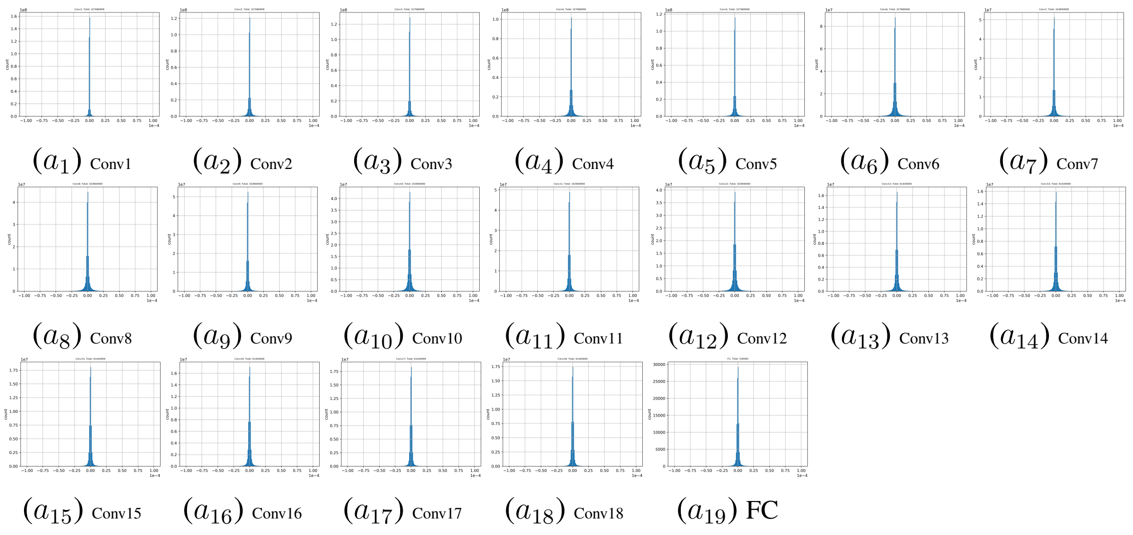


(a_1) Conv1 (a_2) Conv2 (a_3) Conv3 (a_4) Conv4 (a_5) Conv5 (a_6) FC

Figure 2.4: Histograms of relative errors between units directly computed by CNN ($\mathbf{c}_i$) and their values reconstructed by our method ($\mathbf{p}_i$), from Conv1$\sim$5 layers and the FC layer (before ReLU6) of VGG7 (Eq.(2.16), (2.17)). Data were collected from 10k test samples. The x-axis is the error, and the y-axis is the frequency. The total count varies depending on the out-ch feature map size (see table A.1, 3^{rd} column). From (a_1) to (a_6), the percentage of relative errors (Eq.(2.18)) being $\leq 1\%$ are 99.9996%, 99.9996%, 99.9996%, 99.9989%, 99.9989%, 99.996%, respectively.

2.5.2.2 *Visualization of* $\mathbf{RM0}$*:*

$RM0$ works on the FC layer's weights to generate a set of hyperplanes $\{\mathbf{H}_{k}^{Adj}(\mathbf{z}(\mathbf{x}))\}_{k=0}^{9}$ (E-q.(2.9)), one for each final FC layer output unit. Each $\mathbf{H}_{k}^{Adj}(\mathbf{z}(\mathbf{x}))$ represents an effective hyperplane that determines the k^{th} value of the FC output (VGG7: before ReLU6, Fixup-ResNet20: before ReLU19). That is, the VGG7 (or Fixup-ResNet20) taking an image and passing forward


(a_1) Conv1 (a_2) Conv2 (a_3) Conv3 (a_4) Conv4 (a_5) Conv5 (a_6) Conv6 (a_7) Conv7

(a_8) Conv8 (a_9) Conv9 (a_{10}) Conv10 (a_{11}) Conv11 (a_{12}) Conv12 (a_{13}) Conv13 (a_{14}) Conv14

(a_{15}) Conv15 (a_{16}) Conv16 (a_{17}) Conv17 (a_{18}) Conv18 (a_{19}) FC

Figure 2.5: Histograms of relative errors between units directly computed by CNN ($\mathbf{c}_i$) and their values reconstructed by our method ($\mathbf{p}_i$), from Conv1∼18 layers and the FC layer (before ReLU19) of Fixup-ResNet20. The plotting convention is the same as Fig.2.4. The total count varies depending on the out-ch feature map size (see table A.2, 4^{th} column). From (a_1) to (a_{19}), the percentages of relative errors (Eq.(2.18)) being $\leq 1\%$ are greater or equal to 99.997%.

through layers to make a prediction is equivalent to doing ten inner products between the image and our reconstructed hyperplanes $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}))\}_{k=0}^{9}$. Note that each $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}))$ has a shape of $32 \times 32 \times 3$, the same as the input image. We apply the same visualization techniques as $RM4$ (Appendix of [5]).

2.5.2.2.1 Results: We use two images (ship, frog) (same as those used for $RM4$ in the Appendix of [5]). Also, we use an additional image that has the same ship label. The results are shown in Figures 2.6, 2.7. See Section 2.5.3 below for an interpretation of these results.

2.5.3 Interpretation of results

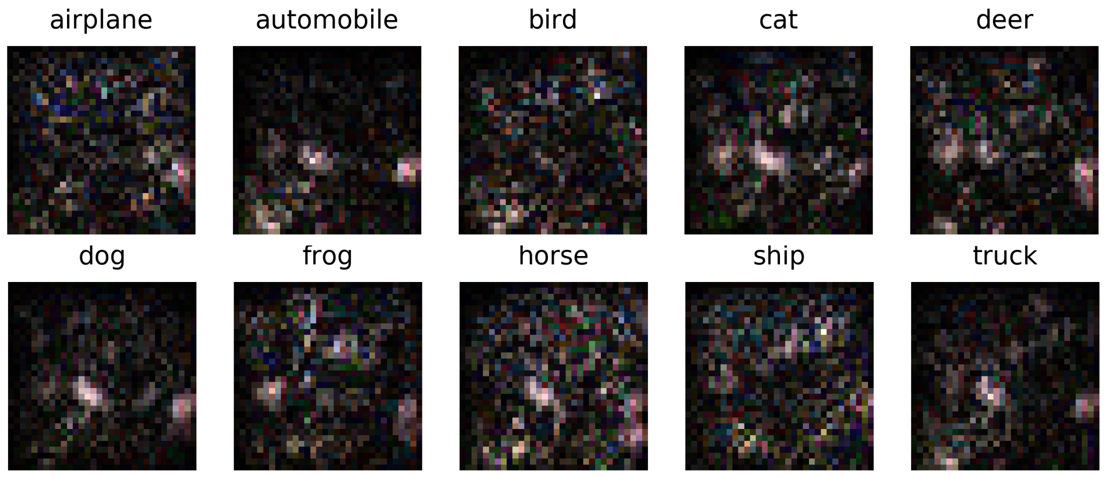
As the layer goes high, the effective receptive field enlarges. (Effective receptive field refers to the receptive field region in the input image that contributes to the unit's computation: see [33] for details.) It is evident from the plots of both $RM4$ and $RM2$ (Appendix of [5]: in each grid cell, the non-black areas mark the effective receptive field). The reason is that if a kernel in Conv0 has a shape of 3×3, the same size kernel in Conv1 actually will have a maximal effective receptive field of 5×5 from the original input perspective because of stride $= 1$, and this trend continues as we go up the hierarchy. However, few kernels fully utilize their effective receptive fields. Some layer has a kernel that takes null from the input image (its effective receptive field blacks out), which makes $\mathbf{H}^{Adj}_{\{s\},j,i}(\mathbf{z}(\mathbf{x}))$ very sparse (eq.(20) in Appendix of [5]).

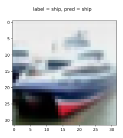
[48] has explored CNN's inner workings in the frequency domain. Several examples in their Figure 2 have illustrated that the high frequency "noise" filtered from an input image can convince CNN of the label while the image's low-frequency content, recognizable for our vision, deceives the CNN. They concluded that CNN might exploit the high-frequency image components which are not perceivable to humans. Our reconstructions of units' decision process observe that effective hyperplanes, regardless of the reconstruction mode ($RM4$ to $RM0$), are not human recognizable patterns in the spatial domain either. For example, patterns of $H^{Adj}_{k=8}(\mathbf{z}(\mathbf{x}))$ ($k = 8$ is the image's label) in Figures 2.6(a,b), 2.7(a, b) show neither a clear ship shape nor a rough contour of a ship. Also, hyperplane patterns of Fig.2.6(a) (or Fig.2.7(a)) are significantly different from Fig.2.6(b) (or Fig.2.7(b)), although they are for the same class (ship) input. We also observe that these hyperplane patterns have strong amplitudes in higher frequency ranges. FFT results, presented in Figures 2.8, 2.9, and 2.10, show that an effective hyperplane has much wider spectra than those in the input image, which implies that CNN indeed takes higher frequency information that may not be perceivable to our humans, confirming the observation in [48]. We further note that $RM3$ and $RM1$ (Appendix of [5]) have object-like colored shapes at low-level convolutional layers. However, they gradually turn to irregular pixels as the layer goes high. Even the same kernel shows different hyperplane patterns dependent on the stride location. This means that a kernel may decide differently from what appears in the input image when it moves to a different receptive field, although the kernel itself does not physically change during the strides. All of the above considered, we can conclude that the CNN's decision is sensitive to values in each input image pixel.

As the proof (Section 2.4) of Eq.(2.3) reveals, changing an input image $\mathbf{x}$ forces $\mathbf{H}^{Adj}(\mathbf{z}(\mathbf{x}))$ to vary at the same time. This has been experimentally tested in Figures 2.4, 2.5. Specifically, given VGG7's or Fixup-ResNet20's entire computation on input image $\mathbf{x}$ denoted as $\mathcal{N}$, for each k^{th} class value, $\mathcal{N}_k(\mathbf{x})$, we have

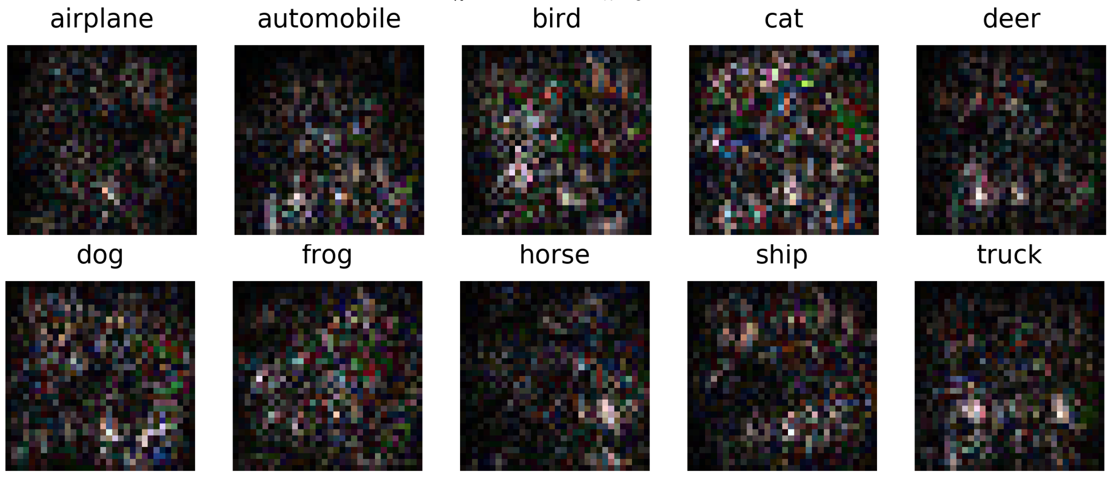
$$\mathcal{N}_k(\mathbf{x}) = \langle \mathbf{x} \mid \mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x})) \rangle. \tag{2.19}$$

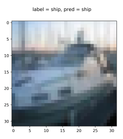
Therefore, the CNN decision process is largely conditioned on the input.



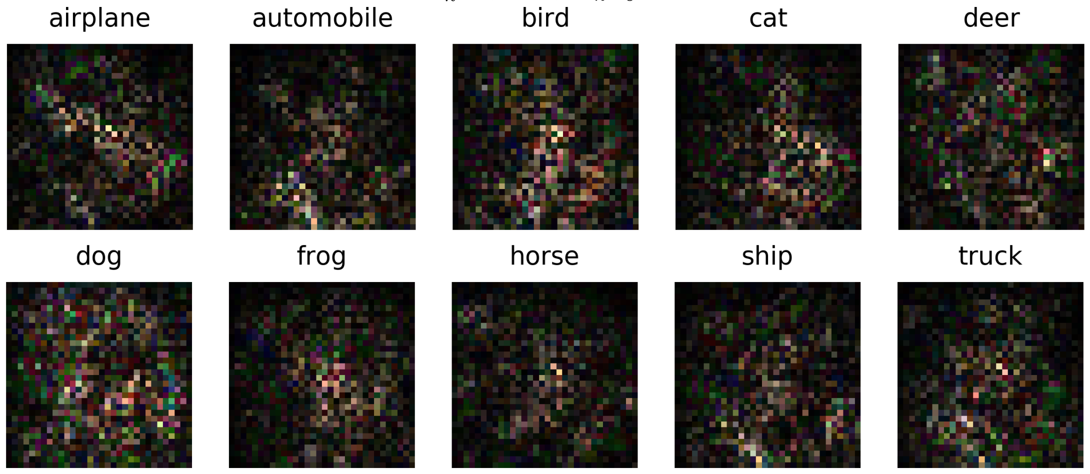


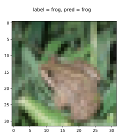
(a) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}_{k=0}^{9}$, VGG7 $(\mathbf{x}_0)$ Ship





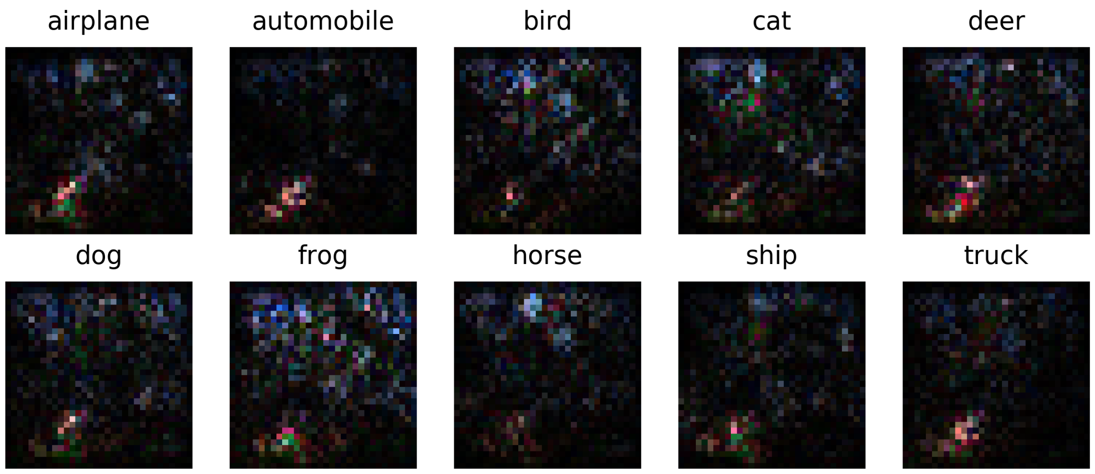
(b) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_1))\}_{k=0}^{9}$, VGG7 $(\mathbf{x}_1)$ Ship



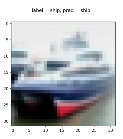


(c) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_2))\}_{k=0}^{9}$, VGG7 $(\mathbf{x}_2)$ Frog

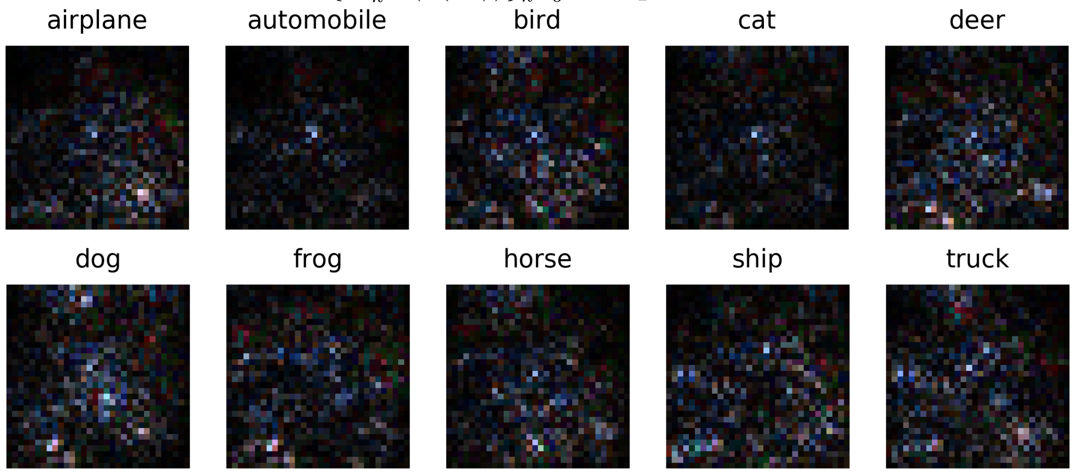
Figure 2.6: Visualization of **RM0** on VGG7. $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_i))$ patterns are mapped from the FC layer (Eq.(2.9)) by $RM0$ (three figures: a, b, c, and the corresponding inputs: $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$). The number of subfigures in a plot equals the number of classes.



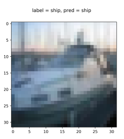
(a) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}_{k=0}^{9}$, Fixup-ResNet20



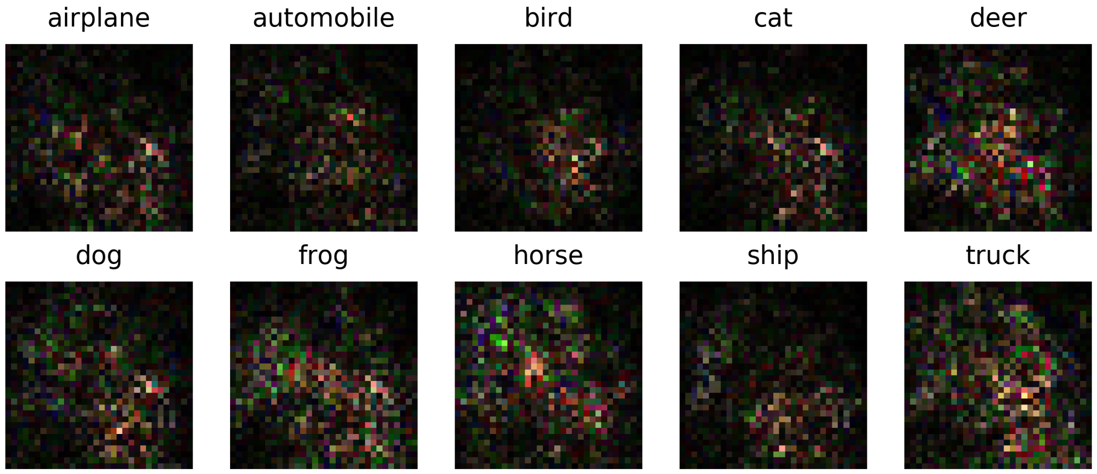
($\mathbf{x}_0$) Ship



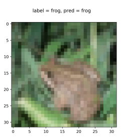
(b) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_1))\}_{k=0}^{9}$, Fixup-ResNet20



($\mathbf{x}_1$) Ship



(c) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_2))\}_{k=0}^{9}$, Fixup-ResNet20



($\mathbf{x}_2$) Frog

Figure 2.7: Visualization of $\mathbf{RM0}$ on Fixup-ResNet20. Details are similar to Fig.2.6.

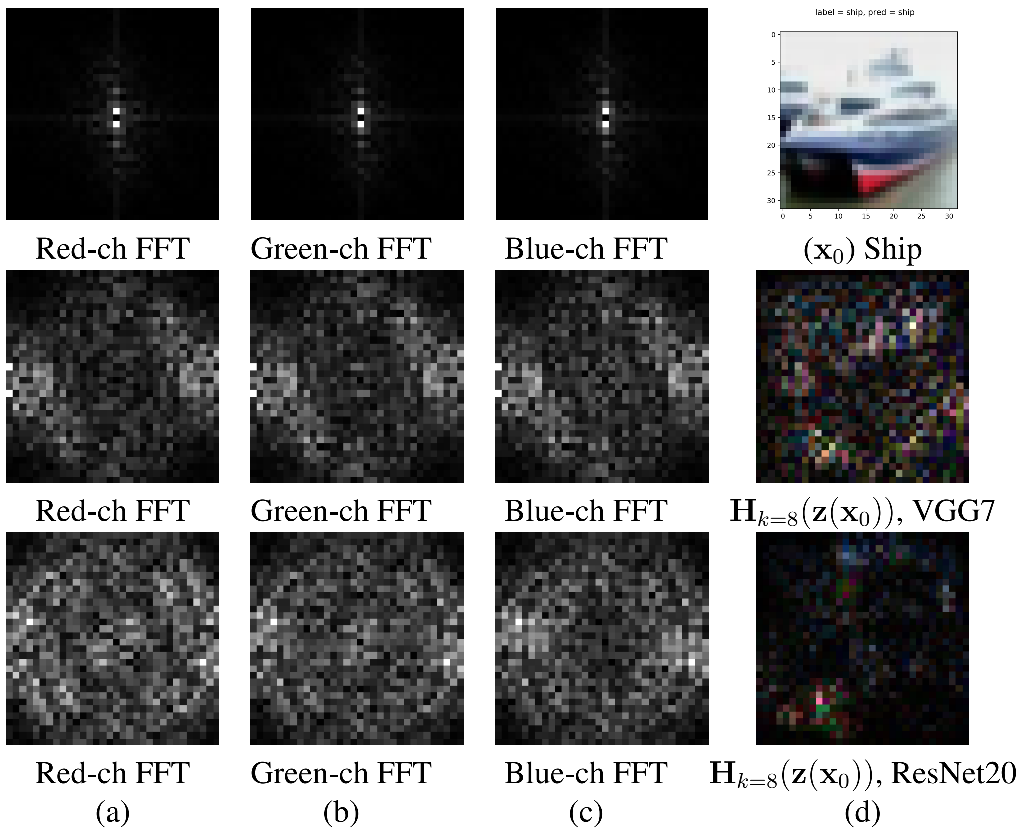


Figure 2.8: Comparison of FFTs of the input image vs. those of the CNNs' effective hyperplanes ($k = 8$, "ship" class). Columns (a)-(c) show the FFTs (Fast Fourier Transform, without the DC component) of the data in column (d). Since both the input image (top row, "Ship") and the hyperplanes (second and third rows, for VGG7 and Fixup-ResNet20) have RGB channels, FFTs of the three separate channels are shown. The ship image's spectrum has a very strong low-frequency component (the first row), while those of the hyperplanes have stronger amplitudes in higher frequency ranges (the second and third rows).

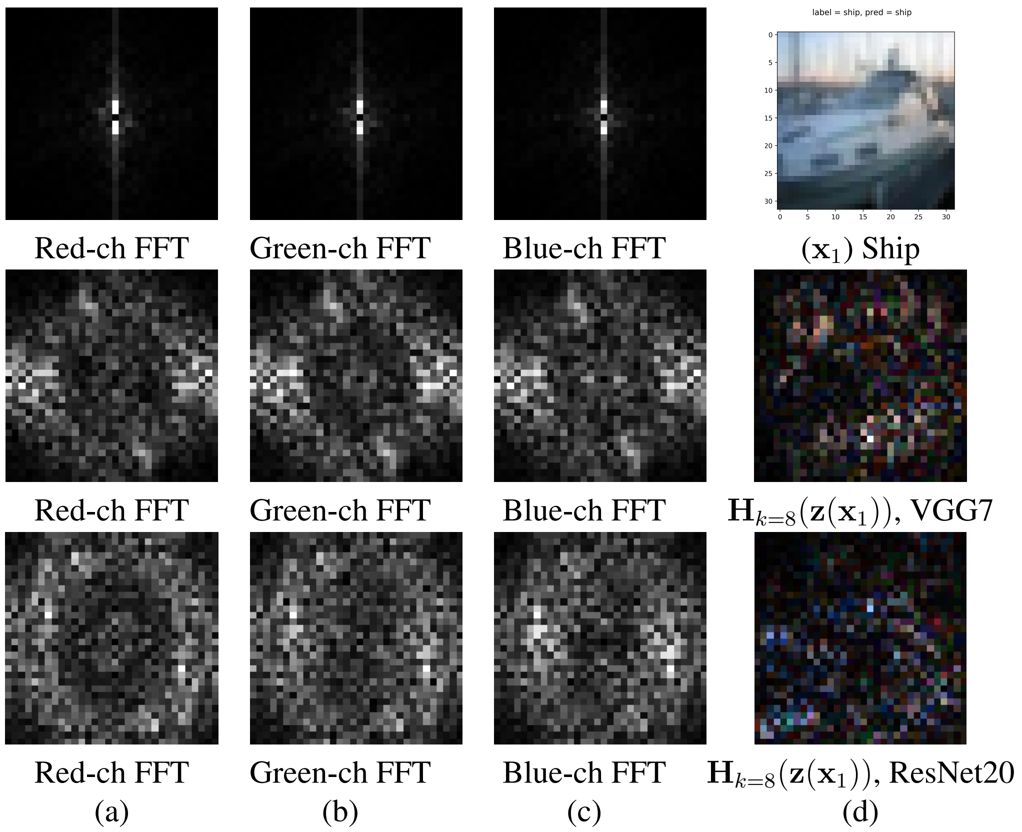


Figure 2.9: Comparison of FFTs of the input image vs. those of the CNNs' effective hyperplanes ($k = 8$, "ship" class). See Fig.2.8 for details.

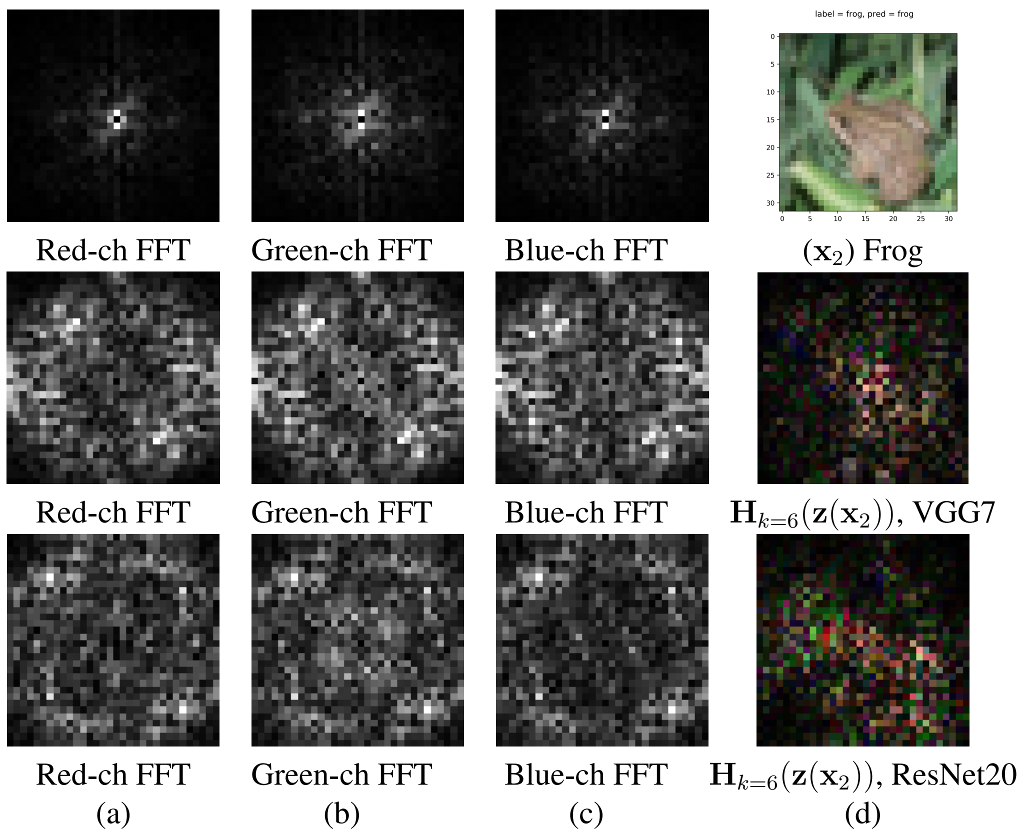


Figure 2.10: Comparison of FFTs of the input image vs. those of the CNNs' effective hyperplanes ($k = 6$, "frog" class). See Fig.2.8 for details.

3. MODELING CNN WITH BIAS

3.1 Overview

Applying adjoint operators to a general CNN is intractable since bias is usually added right after convolution. However, this challenge can be addressed [49] if we treat bias as another input component that multiplies with a weight value $= 1.0$ (Fig.3.1). This way, we preserve the exact computation of the CNN with bias units while maintaining our previous analysis framework for CNN without bias units. This is all fine since we are dealing with an already trained, weight-fixed CNN for analysis only.

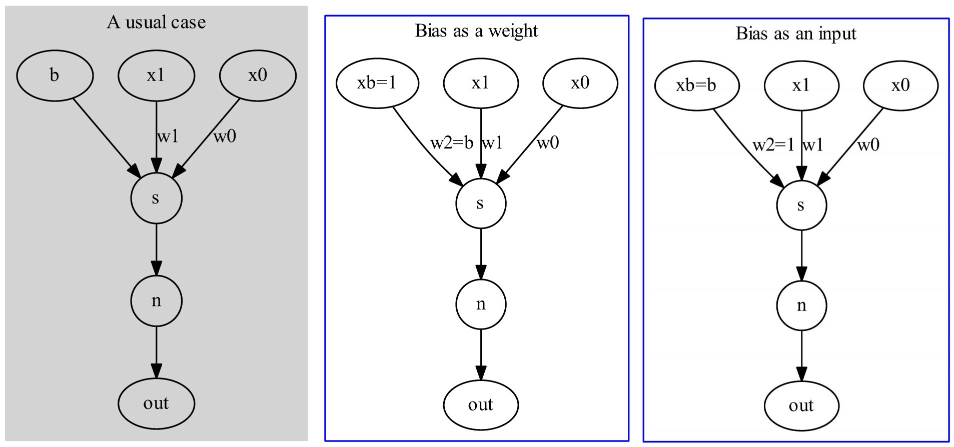


Figure 3.1: Two equivalent models of a trained artificial neuron. **Left**: A conventional artificial neuron with two inputs x_0, x_1, whose weights are w_0, w_1, respectively. Node b denotes a bias. Node s denotes a summation unit, and node n denotes an activation function. **Middle**: An equivalent model to the left. We treat the bias b as the product of the additional input $x_b = 1$, and a weight $w_2 = b$. **Right**: Another equivalent model to the left. We treat bias b as the third input $x_b = b$ but multiplying a weight $w_2 = 1$. Our analysis uses the last model.

3.2 Theory

We will discuss how we model the usual convolution (with bias) in CNN. Although the intuitive idea to map any unit's kernel or weights back to the input space is similar to the one discussed in

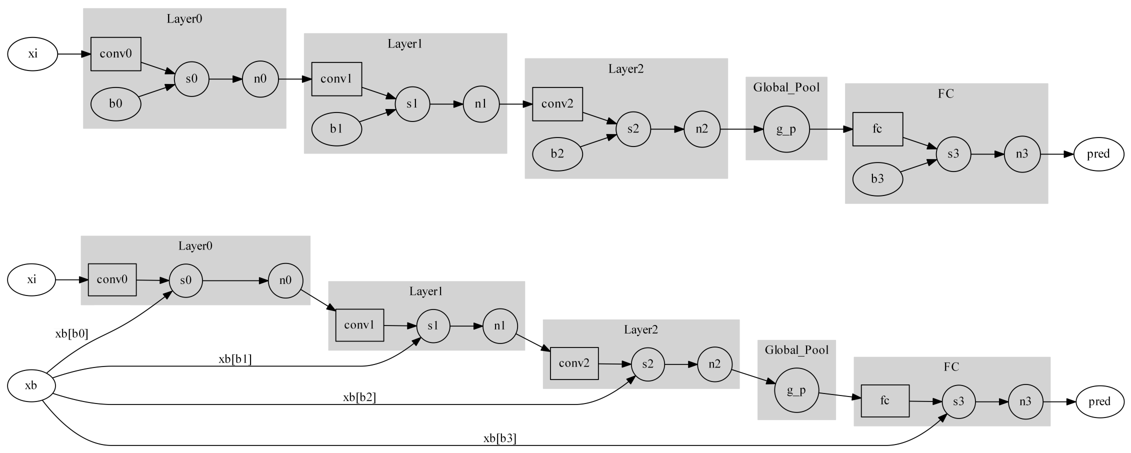


Figure 3.2: An equivalent model of a trained 4-layer CNN (proved in Eq.(3.19), Section 3.4). **Top**: A CNN consists of three Conv layers, one global pooling layer, and one FC layer. Once training is finished, all parameters will be fixed. **Bottom:** We convert the trained CNN to an equivalent model. In this case, the bias values from all layers are sequentially concatenated as a big vector, $\mathbf{x}_b = [\mathbf{b}_0; \mathbf{b}_1; \mathbf{b}_2; \mathbf{b}_3]$. That vector will be fed in as an additional input tensor. Each layer picks its own bias from the $\mathbf{x}_b$ to compute. $\mathbf{x}_b[\mathbf{b}_i]$ denotes that the corresponding part for $\mathbf{b}_i$ will be recovered from the tensor $\mathbf{x}_b$. Note: $\mathbf{x}_b$ is fixed after the CNN is trained, regardless of the input $\mathbf{x}_n$ presented during inference.

Section 2.2, the mathematical scaffold in this section will differ from the previous case due to the incorporated bias term.

Let $\mathcal{X}_I$ and $\mathcal{X}_b$ be two subspaces containing input images and the collected bias values, respectively. We can then construct a normed space $\mathcal{X}$ where $\mathcal{X} = \mathcal{X}_I \bigtimes \mathcal{X}_b$. A point in this space $\mathcal{X}$ would be a concatenation of the input image $\mathbf{x}_n$ and the vector $\mathbf{x}_b$ containing all bias values from the trained CNN: $[\mathbf{x}_n; \mathbf{x}_b]$ (see Fig.3.2). We consider an element-wise inner product in $\mathcal{X}$ (induces the norm on $\mathcal{X}$) defined as,

$$\begin{aligned}\langle[\mathbf{x}_{H\times W\times C}; \mathbf{b}_{L\times 1}] \mid [\mathbf{y}_{H\times W\times C}; \mathbf{d}_{L\times 1}]\rangle &= \sum_{i=0}^{H-1}\sum_{j=0}^{W-1}\sum_{k=0}^{C-1} x_{i,j,k}y_{i,j,k} + \sum_{l=0}^{L-1} b_l d_l \\ &= \langle \mathbf{x}_{H\times W\times C} \mid \mathbf{y}_{H\times W\times C}\rangle + \langle \mathbf{b}_{L\times 1} \mid \mathbf{d}_{L\times 1}\rangle,\end{aligned} \tag{3.1}$$

where H, W, C means the height, width, and color channels of the input image; and L is the total number of bias values in the CNN. The first equivalence is by definition, and we use the second equivalence for tensor implementation.

To follow the description below, please refer to Fig.3.3 as you go along. Suppose $\mathbf{F}$ is the entire computation performed in a CNN's forward path from an input to an $r_1 \times r_2$ receptive field on an in-channel (in-ch) feature map (depicted by a normed space $\mathcal{Y}$); Bias vectors from all layers in the trained CNN are sequentially concatenated into a vector $\mathbf{x}_b$ (Fig.3.2). $\mathbf{F}$ takes the combination of an image $\mathbf{x}_n \in \mathcal{X}_I$ and the vector $\mathbf{x}_b \in \mathcal{X}_b$ as the input. Then, a kernel $\mathbf{w}_{r_1 \times r_2}$ convolving on the receptive field, $\mathbf{F}([\mathbf{x}_n; \mathbf{x}_b])$, before adding the bias vector, can be described in a dual form [42, 43],

$$\begin{aligned} c([\mathbf{x}_n; \mathbf{x}_b]) &= \mathbf{F}([\mathbf{x}_n; \mathbf{x}_b]) \circledast \mathbf{w}_{r_1 \times r_2} \\ = \langle \mathbf{F}([\mathbf{x}_n; \mathbf{x}_b]), \mathbf{w}_{r_1 \times r_2} \rangle &= \langle \mathbf{J}_\mathbf{F}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]))[\mathbf{x}_n; \mathbf{x}_b], \mathbf{w}_{r_1 \times r_2} \rangle, \end{aligned} \tag{3.2}$$

where $c : \mathcal{X} \to \mathbb{R}$ points to a unit in the convolved feature map (before the addition of the bias); $\mathbf{J}_F : \mathcal{X} \to B(\mathcal{X}, \mathcal{Y})$ is a Jacobian operator; The third "=" holds when the CNN is activated with ReLU or Leaky ReLU, and $\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]) = k[\mathbf{x}_n; \mathbf{x}_b]$ for an appropriate $k \in \mathbb{R}^+$ (proved in Section 3.4). Eq.(3.2) implies that $\mathbf{w}_{r_1 \times r_2}$ ($\in \mathcal{Y}^*$) is a local hyperplane on $\mathcal{Y}$ (see [5]).

For a fixed image $\mathbf{x}_0$, and the vector $\mathbf{x}_b$ constructed from a trained CNN, the Adjoint operator $\mathbf{J}^*_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))$ will lead to the following equalities:

$$\begin{aligned} c([\mathbf{x}_0; \mathbf{x}_b]) = \langle \mathbf{J}_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))[\mathbf{x}_0; \mathbf{x}_b], \mathbf{w}_{r_1 \times r_2} \rangle &= \langle [\mathbf{x}_0; \mathbf{x}_b] \mid \mathbf{J}^*_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))\mathbf{w}_{r_1 \times r_2} \rangle \\ &= \langle [\mathbf{x}_0; \mathbf{x}_b] \mid \mathbf{J}^\intercal_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))\mathbf{w}_{r_1 \times r_2} \rangle \\ = \langle \mathbf{x}_0 \mid (\mathbf{J}^\intercal_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))[\mathbf{x}_0])\mathbf{w}_{r_1 \times r_2} \rangle &+ \langle \mathbf{x}_b \mid (\mathbf{J}^\intercal_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))[\mathbf{x}_b])\mathbf{w}_{r_1 \times r_2} \rangle \end{aligned} \tag{3.3}$$

where $\mathbf{J}^\intercal_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))[\mathbf{x}_0], \mathbf{J}^\intercal_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))[\mathbf{x}_b]$ intends to divide the adjoint operator into two parts (two operators) in response to $\mathbf{x}_0$ and $\mathbf{x}_b$; The second "=" and the last "=" holds thanks to the Riesz Representation theorem [43] and the distributive property of a linear operator, respectively. Riesz Representation also contributes to the unification of $\mathcal{X}$ and $\mathcal{X}^*$ or $\mathcal{Y}$ and $\mathcal{Y}^*$. Therefore, we have $\mathbf{w}_{r_1 \times r_2} \in \mathcal{Y}$ and $\mathbf{J}^*_\mathbf{F}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b])) \in B(\mathcal{Y}, \mathcal{X})$. The two operators (Eq.(3.4)) will map a convolution kernel from a convolutional layer or a weight vector from the FC layer all the way back to the image subspace $\mathcal{X}_I$ and the bias subspace $\mathcal{X}_b$, respectively.

$$\begin{aligned} &\mathbf{J}^*_\mathbf{F}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))[\mathbf{x}_0]\mathbf{w}_{r_1\times r_2} \in \mathcal{X}_I, \\ &\mathbf{J}^*_\mathbf{F}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))[\mathbf{x}_b]\mathbf{w}_{r_1\times r_2} \in \mathcal{X}_b. \end{aligned} \tag{3.4}$$

The two will jointly reconstruct an effective hypersurface representing all decision hyperplanes forward from the input to an arbitrary unit of the out-ch feature map or a class value of the FC layer. We name this method AdjointBackMapV2 and observe these three fundamental properties:

1. The mapping, $\mathbf{J}_\mathbf{F}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))$ (i.e., $(\mathbf{J}_\mathbf{F})\circ(\mathbf{z}) : \mathcal{X} \to B(\mathcal{X},\mathcal{Y})$), is not linear since $\exists \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_I$, such that, $\mathbf{J}_\mathbf{F}(\mathbf{z}(\alpha[\mathbf{x}_1,\mathbf{x}_b] + \beta[\mathbf{x}_2,\mathbf{x}_b])) \neq \alpha\mathbf{J}_\mathbf{F}(\mathbf{z}([\mathbf{x}_1,\mathbf{x}_b])) + \beta\mathbf{J}_\mathbf{F}(\mathbf{z}([\mathbf{x}_2,\mathbf{x}_b]))$ for two scalars, α, β;

2. Eq.(3.4) suggests a single effective hyperplane will have two components instead of two separate effective hyperplanes;

3. A connection to the theory of [5] comes from Eq.(3.3) that, if all bias were zeroed out from the CNN's layers, the extended norm space $\mathcal{X}$ would shrink to the $\mathcal{X}_I$ that is identical to the input space discussed in [5]. In other words, AdjointBackMapV2 is more general than our previous theory introuduced in Section 2.2.

3.3 Algorithm

This section will rewrite our previous algorithm (Section 2.3) to fit the more general theory.

3.3.1 Layers considered for analysis

The AdjointBackMapV2 is designed to work on two types of layers inside a CNN. Fig.3.3 illustrates our principles in detail.

1. Any kernel of a convolutional layer (except the kernels from the first layer, which have already been elements in $\mathcal{X}^*$) can be projected back to a joint space $\mathcal{X}$ concatenating the input images and bias vectors. This back-mapped pattern determines the pixel's linear activation value (the weighted sum) of an out-channel (out-ch) feature map;

2. Any weight vector of the FC layer can also be projected back to space $\mathcal{X}$. This back mapped pattern determines the linear activation value of the classification output unit.

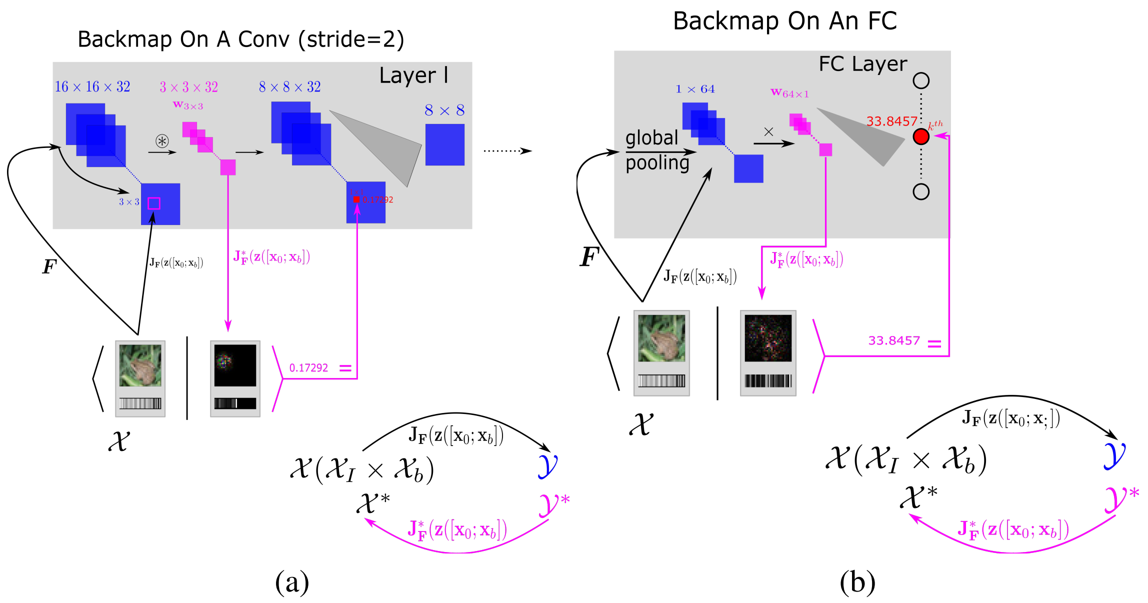


Figure 3.3: Principles of AdjointBackMapV2. Our reconstruction method applies to units in (a) Conv layers, and (b) FC layers. We color elements from the same normed space identically. The normed space $\mathcal{X}$ is an input space that involves two embedded subspaces $\mathcal{X}_I$ and $\mathcal{X}_b$. Considering a fixed image $\mathbf{x}_0 \in \mathcal{X}_I$ and the trained bias values in the CNN (a fixed $\mathbf{x}_b \in \mathcal{X}_b$), $\mathbf{F}$ depicts the CNN's forward computation path from the input end to an $r_1 \times r_2$ receptive field on an in-ch feature map or the global pooling layer. Its adjoint operator, $\mathbf{J}^*_{\mathbf{F}}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))$, projects the corresponding kernel or weights back to the input space $\mathcal{X}$. Riesz Representation unites $\mathcal{X}, \mathcal{X}^*$ or $\mathcal{Y}, \mathcal{Y}^*$, together. The "$\langle \cdot \mid \cdot \rangle$" notation represents a dot-product defined in Eq.(3.1). The symbol "=" means that a dot-product between the extended input and the reconstructed effective hyperplane is equal to the unit's linear activation value (the weighted sum) of a convolved feature map or an FC layer's output value, as long as the CNN is activated with ReLU or Leaky ReLU. See Section 3.2 for a step-by-step walk-through.

3.3.2 Premise

Our algorithm's necessary condition is the third equality ("=") between the left and right-hand sides in Eq.(3.2). Alternatively, any CNN neuron holding Eq.(3.26) should satisfy our requirements. The proof in Section 3.4 reveals that any piecewise linear operations attached to a CNN's layers will not affect the equality. Thus, usual architectural techniques like shortcuts/concatenations/multiple receptive-field kernels' sizes fall under the scope of our analysis. In that case, most variants of CNNs activated with ReLUs or Leaky ReLUs could be analyzed with our method. Our method may not be appropriate for investigating a network activated with functions whose derivatives are not piecewise constants, such as tanh. In general, we insist that numerical precision should be considered when studying a CNN's inner workings (i.e., the method should reproduce the output values precisely), while existing methods of shaping a kernel as a filter did not achieve this.

3.3.3 Incorporating batch normalization

Usually, bias serves CNNs in two operational modes:

1. Acting as trainable parameters attached after convolution operations;
2. Used as moving parameters that track channel-wise average batch values (BN, batch normalization [50]).

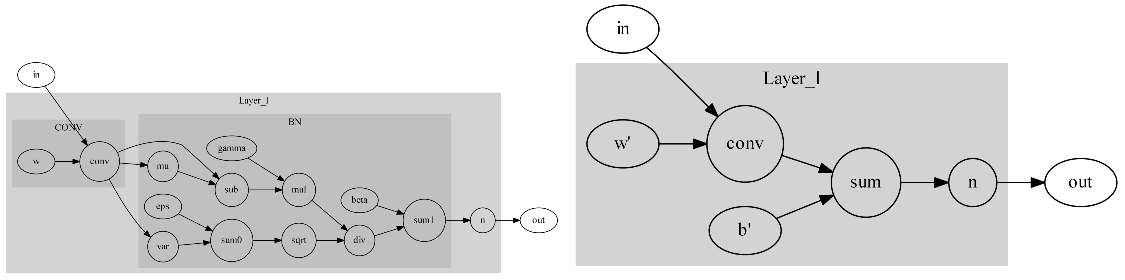
The first one has a similar network topology as Fig.3.2, which is trivial for our theory to accommodate. The batch normalization case is more complex than the first. In the l^{th} layer, the in-channel (in-ch) feature maps $\mathbf{x}_{l-1}$ will convolve with the layer's kernels $\mathbf{w}_l$. The moving mean vector, μ, and moving variance vector, σ^2, will normalize the convolved results before activating the layer's neuron. We summarize the BN computation below.

$$\mathbf{BN}(\mathbf{x}_{l-1}) = \gamma \times \frac{(\mathbf{x}_{l-1} \circledast \mathbf{w}_l - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta, \tag{3.5}$$

where γ, β are two learnable parameters that weight feature maps channel-wise; a preset ϵ prevents any divide by zero exceptions (TensorFlow [45] sets $\epsilon = 0.001$). We can reduce its complexity using Eq.(3.6) (Fig.3.4),

$$\begin{aligned} \mathbf{BN}([\mathbf{x}_{l-1};\mathbf{b}^{'}]) = \mathbf{x}_{l-1} \circledast \mathbf{w}_{l}^{'} + \mathbf{b}^{'} \\ \mathbf{w}_{l}^{'} = \frac{(\gamma \times \mathbf{w}_{l})}{\sqrt{\sigma^{2}+\epsilon}},\ \mathbf{b}' = (\beta - \frac{\gamma \times \mu}{\sqrt{\sigma^{2}+\epsilon}}). \end{aligned} \tag{3.6}$$

It suggests that a CNN trained to use BNs is equivalent to a similar network graph illustrated in Fig.3.2 if $\mathbf{w}_{l}^{'}$ and $\mathbf{b}^{'}$ are extracted from the trained CNN's batch normalization layer. Therefore, applying our theory to a CNN with BNs is as trivial as the first bias operational mode. Besides, Eq.(3.6) significantly reduces both computations and DRAM consumption since fewer multiplications and parameters are required when compared to Eq.(3.5). Note that transformation is possible since we deal with a trained and fixed CNN.



(a) (b)

Figure 3.4: Batch normalization (BN) and its reduction. (a) A standard BN's implementation from Eq.(3.5); (b) Reduction of BN to be compatible with our analysis (similar to Fig.3.2, via Eq.(3.6)). Besides, this transformation lowers the computations and DRAM consumption.

3.3.4 Five reconstruction modes (RMs)

Our AdjointBackMapV2 approach provides five reconstruction modes (RMs) to reconstruct the effective hypersurface, depending on the location of the unit in the CNN, and on the convolution operation's variant: $RM0$ will project from the FC layer, and the remaining four ($RM4$~$RM1$) will project from a convolutional layer. For the four, generally, two factors will distinguish one RM from others:

1. With or without in-ch merge during convolution;
2. With or without global pooling.

All RMs and their algebraic relationships between the forward and backward paths are illustrated in Fig.3.5. Note: although the factors for classifying this section's RMs are similar to those mentioned in [5], their principles are significantly different from [5]'s due to bias being considered.

To help keep track of the steps in the following, we use concrete numerical values. Suppose a CNN with Leaky ReLU as its activation function has been trained on CIFAR-10. It takes a $32 \times 32 \times 3$ (height $\times$ width $\times$ channels) RGB image $\mathbf{x}_n$ to predict 10 classes. Its l^{th} Conv layer has kernels $\mathbf{w}_{l,3\times3\times32\times64}$ (h $\times$ w $\times$ in-chs $\times$ out-chs) convolving on $16\times16\times32$ in-ch feature maps with convolution stride $= 2$ and padding = "SAME" [45]. The implementation of 2-D convolution [45] states: a kernel $\mathbf{w}_{3\times3}$ only convolves on its corresponding 16×16 in-ch feature map through 8×8 (a stride move s ranges from 0 to 63). Then, all 32 convolved feature maps are added together in-channel-wise (in-ch summation) as an 8×8 out-ch feature map that will be subsequently added to the bias. Thus, the l^{th} layer is supposed to produce an $8 \times 8 \times 64$ out-ch feature map. Besides, it may have global pooling [4] (g_p) applied right after the activated feature maps, before entering the FC layer.

3.3.4.1 $RM4$ (Fig.3.5(a))

Neither merging nor global pooling is performed on either in-ch kernels or training strides. In this case, a kernel will be individually mapped via $\mathbf{H}^{Adj}_{l,s,j,i}$, composed of $\mathbf{H}^{Adj,I}_{l,s,j,i}$ and $\mathbf{H}^{Adj,b}_{l,s,j,i}$, to space $\mathcal{X}$, defined as:

$$\begin{aligned}\mathbf{H}^{Adj,I}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= (\mathbf{J}^*_{\mathbf{F}_{l-1,s,j,i}}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))[\mathbf{x}_n])\mathbf{w}_{l,3\times3,s,j,i},\\ \mathbf{H}^{Adj,b}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= (\mathbf{J}^*_{\mathbf{F}_{l-1,s,j,i}}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))[\mathbf{x}_b])\mathbf{w}_{l,3\times3,s,j,i}\\ s \in \{0,1,...,63\}, &\, j \in \{0,1,..,31\}, i \in \{0,1,...,63\},\end{aligned} \tag{3.7}$$

where $\mathbf{F}_{l-1,s,j,i}$ denotes the forward path from the input end to an in-ch 3×3 receptive field that will be convolved to the (j,i) out-ch unit at the stride move s. $\mathbf{H}^{Adj}_{l,s,j,i}$ reflects how the local kernel behaves on the combined image and bias input when the stride moves. $\langle [\mathbf{x}_n; \mathbf{x}_b] \mid [\mathbf{H}^{Adj,I}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])); \mathbf{H}^{Adj,b}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]))] \rangle$ should be equal to the unit's value before the in-ch summation. Fig.3.3(a) shows the details as well.

3.3.4.2 $RM3$ (Fig.3.5(b))

No merging is taken on any in-ch kernel's convolution. But, global pooling will be applied for mapping, i.e., the effective hypersurfaces from an individual kernel sum together pixel-wise to reconstruct an effective hypersurface, $\mathbf{H}^{Adj}_{l,j,i}$. That effective hypersurface describes the local kernel's weighting on the input space, considering all stride moves merged. In other words, it reveals how a feature sum could be generated from the space $\mathcal{X}$'s perspective, when an in-channel convolved feature map is pooling globally, i.e.,

$$\begin{aligned}\sum_{s=0}^{63} \mathbf{F}_{l-1,s,j,i}([\mathbf{x}_n, \mathbf{x}_b]) \circledast \mathbf{w}_{l,3\times3,s,j,i} &= \langle [\mathbf{x}_n; \mathbf{x}_b], \mathbf{H}^{Adj}_{l,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])) \rangle \\ &= \langle \mathbf{x}_n, \sum_{s=0}^{63} (\mathbf{J}^*_{\mathbf{F}_{l-1,s,j,i}}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]))[\mathbf{x}_n]) \mathbf{w}_{l,3\times3,s,j,i} \rangle \\ &\quad + \langle \mathbf{x}_b, \sum_{s=0}^{63} (\mathbf{J}^*_{\mathbf{F}_{l-1,s,j,i}}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]))[\mathbf{x}_b]) \mathbf{w}_{l,3\times3,s,j,i} \rangle,\end{aligned} \tag{3.8}$$

where the distributive law and linearity in dual space support the last "=". Thus, $\mathbf{H}^{Adj}_{l,j,i}$ is composed of two operators: $\mathbf{H}^{Adj,I}_{l,j,i}$ and $\mathbf{H}^{Adj,b}_{l,j,i}$,

$$\begin{aligned}\mathbf{H}^{Adj,I}_{l,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])) &= \sum_{s=0}^{63} \mathbf{H}^{Adj,I}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])), \\ \mathbf{H}^{Adj,b}_{l,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])) &= \sum_{s=0}^{63} \mathbf{H}^{Adj,b}_{l,s,j,i}(\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])).\end{aligned} \tag{3.9}$$

The relationship to $RM4$ is evident from Eq.(3.9) (i.e., summation of Eq.(3.7) over s).

3.3.4.3 $RM2$ (Fig.3.5(c))

We do not apply the global pooling. Instead, the kernels' back maps will merge in-channel-wise to reconstruct an effective hypersurface $\mathbf{H}^{Adj}_{l,s,i}$ that determines the unit's linear activation value of an out-ch feature map. Its two operators, $\mathbf{H}_{l,s,i}^{Adj,I}$ and $\mathbf{H}_{l,s,i}^{Adj,b}$, are defined below,

$$\begin{aligned}\mathbf{H}_{l,s,i}^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= \sum_{j=0}^{31}\mathbf{H}_{l,s,j,i}^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])),\\ \mathbf{H}_{l,s,i}^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= \sum_{j=0}^{31}\mathbf{H}_{l,s,j,i}^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])).\end{aligned} \tag{3.10}$$

Eq.(3.10) also relates $RM2$ to $RM4$ (i.e., summation of Eq.(3.7) over j).

3.3.4.4 $RM1$ (Fig.3.5(d))

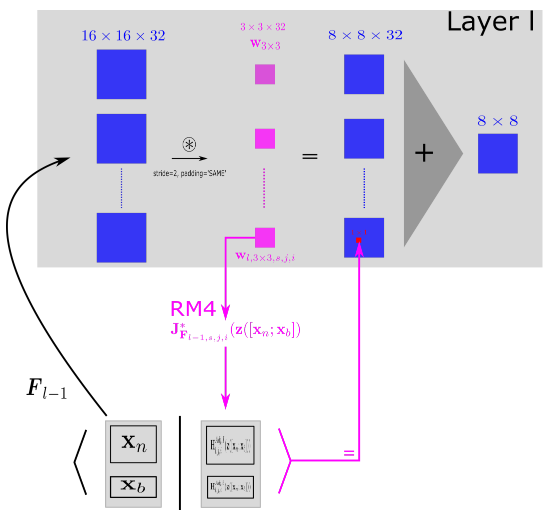
We map with both merging and global pooling considered. Then, a reconstruction will be conducted using $\mathbf{H}_{l,i}^{Adj}$ that determines the linear activation value of the feature maps for the global-pooling layer. Its two parts, $\mathbf{H}_{l,i}^{Adj,I}$ and $\mathbf{H}_{l,i}^{Adj,b}$, and their relationships to $RM3$, $RM2$ are summarized below.

$$\begin{aligned}\mathbf{H}_{l,i}^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= \sum_{j=0}^{31}\mathbf{H}_{l,j,i}^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) = \sum_{s=0}^{63}\mathbf{H}_{l,s,i}^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))\\ \mathbf{H}_{l,i}^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= \sum_{j=0}^{31}\mathbf{H}_{l,j,i}^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) = \sum_{s=0}^{63}\mathbf{H}_{l,s,i}^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])).\end{aligned} \tag{3.11}$$

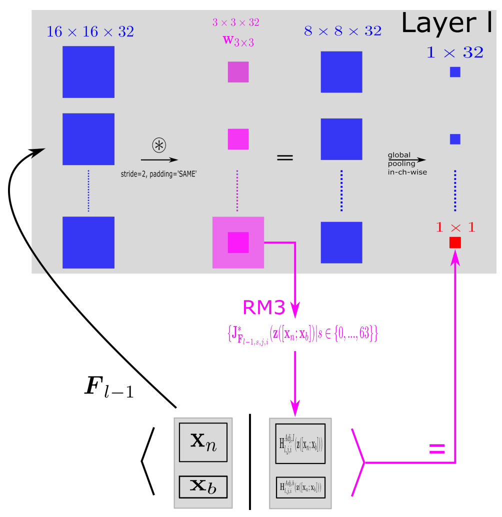
3.3.4.5 $RM0$ (Fig.3.3(b))

Mapping an FC weight vector $\mathbf{w}_k$ is independent of the factors that govern the convolution operation. An effective hypersurface $\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))$ reconstructed in this way represents the whole decision process towards a predicted value for the k^{th} class. $\mathbf{H}_k^{Adj}$ consists of two operators $\mathbf{H}_k^{Adj,I}$ and $\mathbf{H}_k^{Adj,b}$ ($k \in \{0,1,...,9\}$) as well.

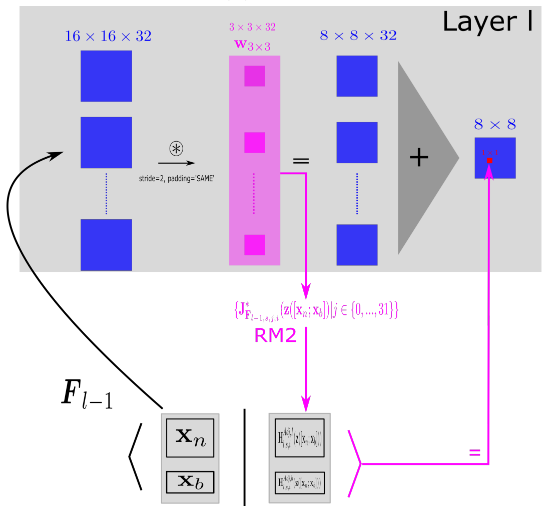
$$\begin{aligned}\mathbf{H}_k^{Adj,I}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= (\mathbf{J}_{\mathbf{F}_{g\_p}}^{*}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))[\mathbf{x}_n])\mathbf{w}_k,\\ \mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b])) &= (\mathbf{J}_{\mathbf{F}_{g\_p}}^{*}(\mathbf{z}([\mathbf{x}_n;\mathbf{x}_b]))[\mathbf{x}_b])\mathbf{w}_k.\end{aligned} \tag{3.12}$$



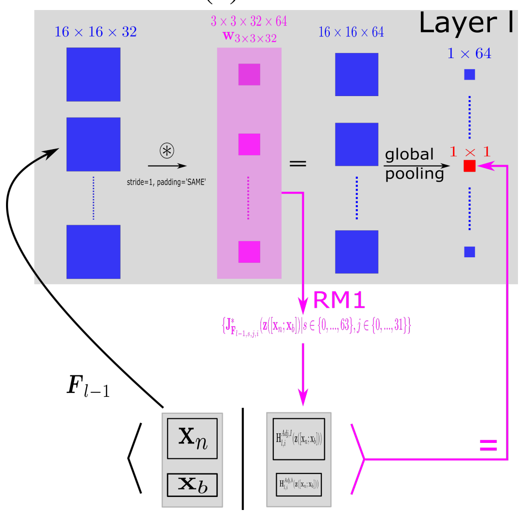
(a) $RM4$



(b) $RM3$



(c) $RM2$



(d) $RM1$

Figure 3.5: Two factors (in-ch merge & global pooling (g_p)) determine four RMs involved in the Conv layers. (a) $RM4$: Without either in-ch merge or g_p; (b) $RM3$: Without in-ch merge and with g_p; (c) $RM2$: With in-ch merge but without g_p; (d) $RM1$: With both in-ch merge and g_p. The usage of colors is similar to Fig.3.3. An oversized pink mask in either (b) or (d) denotes an effective hypersurface reconstructed in response to a stride-wise summation of the convolved feature map. The symbol "$\langle \cdot \mid \cdot \rangle$" refers to the inner product defined in Eq.(3.1).

3.3.5 Implementation

Computing a jacobian is expensive, and we still use convolution to accelerate our effective hypersurface reconstruction. Since we consider bias this time, the computational procedures are different from the previous one (Algorithm 1). Eq.(3.7)~(3.12) are optimized and compiled in

Algorithm 2. The paddings should be identical to those used during training. Tensorflow [45] functions matmul, unstack, stack, expanddim, conv2d, sum, are used in the algorithm. The transpose in Eq.(3.3) is achieved via an 'axis' option of the conv2d. From the algorithm we can see how the different reconstruction modes (RMs) are related.

3.4 Proof of Eq.(3.2)

We show $\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]) = k[\mathbf{x}_n; \mathbf{x}_b] (k \in \mathbb{R}^+)$ achieves the third equality ("=") in Eq.(3.2) if the units in the CNN are activated with ReLU or Leaky ReLU activation function. We also generalize this conclusion to any piecewise linear activation function when $k = 1$ is applied.

3.4.1 Notations and concepts

Let $\mathbf{N} : \mathcal{X}_I \to \mathbb{R}^K$ be a CNN (without the last softmax layer) with L convolutional layers, and $\mathbb{R}^{H\times W\times C}$ be an instance of input image space $\mathcal{X}_I$ where H, W, C denote height, width, color channels of an image. Suppose $\mathbf{x}_n \in \mathbb{R}^{H\times W\times C}$ is an input image; The l^{th} layer has convolutional kernels $\mathbf{w}_{r_{l,1}\times r_{l,2}\times c_{l,in}\times c_{l,out}}$ and bias $\mathbf{b}_l$ $(0 \le l \le L-1)$. We use $\mathbf{c}_l(\mathbf{x}_n)$ to represent the convolved feature maps with a bias vector added in the l^{th} layer, and $\hat{\mathbf{c}}_l(\mathbf{x}_n)$ the vectorization of $\mathbf{c}_l(\mathbf{x}_n)$, where $\hat{\cdot}$ is a vectorization operator. We use $\mathbf{p}_l(\mathbf{x}_n)$ to represent the activated feature maps in the l^{th} layer (the layer has an activation σ_l)), and $\hat{\mathbf{p}}_l(\mathbf{x}_n)$ the vectorization of $\mathbf{p}_l(\mathbf{x}_n)$. Their shapes are $\mathbf{c}_l, \mathbf{p}_l \in \mathbb{R}^{h_l\times w_l\times c_{l,in}}$ and $\hat{\mathbf{c}}_l, \hat{\mathbf{p}}_l, \mathbf{b}_l \in \mathbb{R}^{m_l}$ where $m_l = h_l w_l c_{l,in}$. Specifically, $(h_{-1}, w_{-1}, c_{-1,in}) = (H, W, C)$, $m_{-1} = HWC$, and $\hat{\mathbf{p}}_{-1}(\mathbf{x}_n) = \hat{\mathbf{c}}_{-1}(\mathbf{x}_n) = \hat{\mathbf{x}}_i \in \mathbb{R}^{m_{-1}}$. Then, their relationships can be described as below:

$$\begin{aligned} \mathbf{c}_l(\mathbf{x}_n) &= \mathbf{p}_{l-1}(\mathbf{x}_n) \circledast \mathbf{w}_{r_{l,1}\times r_{l,2}\times c_{l,in}\times c_{l,out}} + \mathbf{b}_l \\ &= \hat{\mathbf{c}}_l(\mathbf{x}_n) = \mathbf{A}_l(\hat{\mathbf{p}}_{l-1}(\mathbf{x}_n)) = \mathbf{W}_l(\hat{\mathbf{p}}_{l-1}(\mathbf{x}_n)) + \mathbf{b}_l, \\ \mathbf{p}_l(\mathbf{x}_n) &= \sigma_l(\mathbf{c}_l(\mathbf{x}_n)) = \sigma_l(\hat{\mathbf{c}}_l(\mathbf{x}_n)) = \hat{\mathbf{p}}_l(\mathbf{x}_n), \end{aligned} \tag{3.13}$$

where $\mathbf{W}_l \in \mathbb{R}^{m_t\times m_{t-1}}$ is the matrix representation for the layer's convolution, and $\mathbf{A}_l$ is an affine operator that describes the convolution and bias addition. The neural network parameters, $\{(\mathbf{W}_l, \mathbf{b}_l)\}_{l=0}^{l=L-1}$ defines the composition of convolutions, average/max poolings (average pooling

Algorithm 2: AdjointBackMapV2 with five modes (RM_4 to RM_0)

```
Input: 1. $\mathbf{x}$: input image (shape: $d_1 = H \times W \times C$); 2. $\mathbf{b}$: input bias vector concatenated
       from a trained model (shape: $d_2$); 3. $\mathbf{z}$: the function for Eq.(3.2); 4. $\mathbf{F}_l$: a forward
       mapping to in channels of the $l^{th}$ layer; 5. $\mathbf{w}_l$: the weights at the $l^{th}$ layer; 6. $s$: the
       convolution stride used during training; 7. $M$: RM to be used.
Output: Two parts of an effective hypersurface: $\mathbf{H}^{Adj,I}(\mathbf{z}([\mathbf{x};\mathbf{b}])), \mathbf{H}^{Adj,b}(\mathbf{z}([\mathbf{x};\mathbf{b}]))$
Function AdjointBackMapV2($\mathbf{x}, \mathbf{b}, \mathbf{z}, \mathbf{F}, \mathbf{w}_l, s, L$):
    $\mathbf{z}_0 = \mathbf{z}([\mathbf{x};\mathbf{b}])$
    if $M$ is '$RM_0$' then                                        // 1.  FC layer
        $\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times c_{g\_p,in}}, \mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times c_{g\_p,in}} = \frac{\partial \mathbf{F}_{c_{g\_p,in}}}{\partial \mathbf{x}}, \frac{\partial \mathbf{F}_{c_{g\_p,in}}}{\partial \mathbf{b}}$
        return matmul($\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times c_{g\_p,in}}(\mathbf{z_0})$, $\mathbf{w}_{fc,c_{g\_p,in} \times c_{labels}}$, axis='$c_{g\_p,in}$'),
         matmul($\mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times c_{g\_p,in}}(\mathbf{z_0})$, $\mathbf{w}_{fc,c_{g\_p,in} \times c_{labels}}$, axis='$c_{g\_p,in}$')
    $\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times h_l \times w_l \times c_{l,in}}, \mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times h_l \times w_l \times c_{l,in}} = \frac{\partial \mathbf{F}_{h_l \times w_l \times c_{l,in}}}{\partial \mathbf{x}}, \frac{\partial \mathbf{F}_{h_l \times w_l \times c_{l,in}}}{\partial \mathbf{b}}$
    switch $M$ do
        case '$RM_4$' or '$RM_3$' do                         // 2.  Without in-ch merge
            $\mathbf{w}_{r_1 \times r_2 \times c_{l,out}}$ = unstack($\mathbf{w}_{l,r_1 \times r_2 \times c_{l,in} \times c_{l,out}}$, axis='$c_{l,in}$')
            $\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times h_l \times w_l}$ = unstack($\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times h_l \times w_l \times c_{l,in}}$, axis='$c_{l,in}$')
            $\mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times h_l \times w_l}$ = unstack($\mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times h_l \times w_l \times c_{l,in}}$, axis='$c_{l,in}$')
            $j = 0$, Empty container $R_x$, $R_b$
            while $j < c_{l,in}$ do
                $\mathbf{J}_{\mathbf{F},\mathbf{x}}$ = expanddim($\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times h_l \times w_l}[j]$, axis='$c_{l,in}$')
                $\mathbf{J}_{\mathbf{F},\mathbf{b}}$ = expanddim($\mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times h_l \times w_l}[j]$, axis='$c_{l,in}$')
                $\mathbf{w}$ = expanddim($\mathbf{w}_{r_1 \times r_2 \times c_{l,out}}[j]$, axis='$c_{l,in}$')
                $R_x$.append(conv2d($\mathbf{J}_{\mathbf{F},\mathbf{x}}(\mathbf{z}_0)$, $\mathbf{w}$, stride=$s$, axis='$(h_l, w_l, c_{l,in}, c_{l,out})$'))
                $R_b$.append(conv2d($\mathbf{J}_{\mathbf{F},\mathbf{b}}(\mathbf{z}_0)$, $\mathbf{w}$, stride=$s$, axis='$(h_l, w_l, c_{l,in}, c_{l,out})$'))
                $j = j + 1$
            $\mathbf{H}_I, \mathbf{H}_b$ = stack($R_x$, axis='$c_{l,in}$'), stack($R_b$, axis='$c_{l,in}$')
            if $M$ is '$RM_4$' then                              // 2.1 without g_p
                return $\mathbf{H}_I$, $\mathbf{H}_b$
            else if $M$ is '$RM_3$' then                           // 2.2 with g_p
                return sum($\mathbf{H}_I$, axis='$(h_{l,o}, w_{l,o})$'), sum($\mathbf{H}_b$, axis='$(h_{l,o}, w_{l,o})$')
        case '$RM_2$' or '$RM_1$' do                            // 3.  With in-ch merge
            $\mathbf{H}_I$ = conv2d($\mathbf{J}_{\mathbf{F},\mathbf{x},d_1 \times h_l \times w_l \times c_{l,in}}(\mathbf{z}_0)$, $\mathbf{w}_{l,r_1 \times r_2 \times c_{l,in} \times c_{l,out}}$, stride=$s$,
             axis='$(h_l, w_l, c_{l,in}, c_{l,out})$')
            $\mathbf{H}_b$ = conv2d($\mathbf{J}_{\mathbf{F},\mathbf{b},d_2 \times h_l \times w_l \times c_{l,in}}(\mathbf{z}_0)$, $\mathbf{w}_{l,r_1 \times r_2 \times c_{l,in} \times c_{l,out}}$, stride=$s$,
             axis='$(h_l, w_l, c_{l,in}, c_{l,out})$')
            if $M$ = '$RM_2$' then                               // 3.1 without g_p
                return $\mathbf{H}_I$, $\mathbf{H}_b$
            else if $M$ = '$RM_1$' then                            // 3.2 with g_p
                return sum($\mathbf{H}_I$, axis='$(h_{l,o}, w_{l,o})$'), sum($\mathbf{H}_b$, axis='$(h_{l,o}, w_{l,o})$')
    return NULL
```

is equivalent to convolving with $r_{l,1} \times r_{l,2}$-sized averaging kernels, and max pooling is equivalent to convolving with $r_{l,1} \times r_{l,2}$-sized one-hot kernels), and batch normalizations inside the CNN $\mathbf{N}$. We formalize the pixel-wise activation function σ_l as following,

$$\sigma_l(c) = max(c, 0) + \gamma_l min(c, 0), \forall c \in \mathbb{R}, \tag{3.14}$$

where the leakiness $\gamma_l = 0$ implies σ_l being ReLU, and $0 < \gamma_l < 1$ implies σ_l being Leaky ReLU. The convolutional layers are terminated at $l = L - 1$, and the overall final output of the CNN $\mathbf{N}(\mathbf{x}_l) \in \mathbb{R}^K$ is given by,

$$\begin{aligned} \mathbf{c}_L(\mathbf{x}_n) = \hat{\mathbf{c}}_L(\mathbf{x}_n) = \mathbf{A}_L(\hat{\mathbf{c}}_{L-1}(\mathbf{x}_n)) = \mathbf{W}_L\hat{\mathbf{c}}_{L-1}(\mathbf{x}_n) + \mathbf{b}_L \\ \mathbf{N}(\mathbf{x}_n) = \hat{\mathbf{p}}_L(\mathbf{x}_n) = \sigma_L(\hat{\mathbf{c}}_L(\mathbf{x}_n)). \end{aligned} \tag{3.15}$$

Combining the above, we have ($\circ$ denotes the operator composition),

$$\mathbf{N}(\mathbf{x}_n) = (\sigma_L \circ \mathbf{A}_L \circ \sigma_{L-1} \circ \mathbf{A}_{L-1} \circ ...\sigma_1 \circ \mathbf{A}_1 \circ \sigma_0 \circ \mathbf{A}_0)(\hat{\mathbf{p}}_{-1}(\mathbf{x}_n)). \tag{3.16}$$

3.4.2 Representation of the equivalent topology in Fig.3.2

We use $M = \sum_{l=0}^{L} m_l$, which is equal to $\mathbf{x}_b$'s dimensions. For $0 \leq l \leq L$, we introduce a restriction operator $\mathbf{r}_l : \mathbb{R}^M \to \mathbb{R}^{m_l}$, by

$$\mathbf{r}_l(\mathbf{v}_b) = \mathbf{y}_l, \forall \mathbf{v}_b = [\mathbf{y}_0; \mathbf{y}_1; ...; \mathbf{y}_L] \in \mathbb{R}^M \text{ with } \mathbf{y}_l \in \mathbb{R}^{m_l}. \tag{3.17}$$

Note that $\mathbf{v}_b$ is a variable vector who has an instance $\mathbf{x}_b$. Then, we define a sequence of auxiliary mappings $\{\mathscr{C}_l\}_{l=0}^{l=L-1}$ by the recurrence relation: for $[\hat{\mathbf{x}}_i; \mathbf{v}_b] \in \mathbb{R}^{m_{-1}} \bigtimes \mathbb{R}^M$,

$$\begin{aligned} \mathscr{C}_0([\hat{\mathbf{x}}_i; \mathbf{v}_b]) = \mathbf{W}_0\hat{\mathbf{x}}_i + \mathbf{r}_0(\mathbf{v}_b), \\ \mathscr{C}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]) = \mathbf{W}_l(\sigma_{l-1}(\mathscr{C}_{l-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]))) + \mathbf{r}_l(\mathbf{v}_b), 0 \leq l \leq L-1, \\ \mathscr{C}_L([\hat{\mathbf{x}}_i; \mathbf{v}_b]) = \mathbf{W}_L(\sigma_{L-1}(\mathscr{C}_{L-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]))) + \mathbf{r}_L(\mathbf{v}_b). \end{aligned} \tag{3.18}$$

Combining Eq.(3.13) and (3.15), we get:

$$\mathbf{c}_l(\mathbf{x}_n) = \hat{\mathbf{c}}_l(\mathbf{x}_n) = \mathscr{C}_l([\hat{\mathbf{x}}_i; \mathbf{x}_b]), \tag{3.19}$$

where $\mathbf{v}_b$ is replaced by $\mathbf{x}_b$.

3.4.3 Proof of Eq.(3.2)

From Eq.(3.17), for any $[\hat{\mathbf{x}}_i; \mathbf{v}_b] \in \mathbb{R}^{m_{-1}} \bigtimes \mathbb{R}^M$, we define a sequence of matrices $\{\mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b])\}_{l=0}^{L}$by the recurrence relation:

$$\begin{aligned} \mathbf{J}_0([\hat{\mathbf{x}}_i; \mathbf{v}_b]) &= [\mathbf{W}_0, \mathbf{R}_0] \in \mathbb{R}^{m_0 \times (m_{-1}+M)}, \\ \mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]) &= \mathbf{W}_l \mathbf{\Sigma}_{l-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]) \mathbf{J}_{l-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]) + [\mathbf{O}_l, \mathbf{R}_l], 0 \le l \le L-1 \\ \mathbf{J}_L([\hat{\mathbf{x}}_i; \mathbf{v}_b]) &= \mathbf{W}_L \mathbf{\Sigma}_{L-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]) \mathbf{J}_{L-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b]) + [\mathbf{O}_L, \mathbf{R}_L], \end{aligned} \tag{3.20}$$

where $[\cdot, \cdot]$ is an operator that concatenates two matrices; $\mathbf{R}_l \in \mathbb{R}^{m_t \times M}$ for $0 \le l \le L-1$, and $\mathbf{R}_L \in \mathbb{R}^{k \times M}$, is the matrix representations of the restriction operator $\mathbf{r}_l$; $\mathbf{O}_l \in \mathbb{R}^{m_t \times m_{-1}}$ is a zero matrix; $\mathbf{\Sigma}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]) \in \mathbb{R}^{m_t \times m_t}$ is the Jacobian matrix of ReLU or Leaky ReLU in the l^{th} convolutional layer, which is precisely given by,

$$\mathbf{\Sigma}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]) = diag\left(\frac{1+\gamma_l}{2} + \frac{1-\gamma_l}{2} sgn(\mathscr{C}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]))\right). \tag{3.21}$$

In fact, $\mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b])$ is the Jacobian matrix of $\mathscr{C}_l$ at $[\hat{\mathbf{x}}_i; \mathbf{v}_b]$. We point out that Eq.(3.21) performs a hierarchal separation of the domain in the sense that, given a sequence of binary vectors $\mathbf{e} = \{e_l\}_{l=0}^{L}$ with $e_l \in \{-1, 1\}^{m_l}$ for $0 \le l \le L$, the Jacobian matrix $\mathbf{J}_l$ shares on the same value on the subdomain $\Omega_l(\mathbf{e})$, where

$$\begin{aligned} \Omega_0(\mathbf{e}) &= \mathbb{R}^{m_{-1}} \bigtimes \mathbb{R}^M, \\ \Omega_l(\mathbf{e}) &= \{[\hat{\mathbf{x}}_i; \mathbf{v}_b] \in \Omega_{l-1}(\mathbf{e}) : sgn(\mathscr{C}_{l-1}([\hat{\mathbf{x}}_i; \mathbf{v}_b])) = e_l\}, 0 \le l \le L. \end{aligned} \tag{3.22}$$

As a direct consequence, for any $0 \leq l \leq L$ and $k \in \mathbb{R}^+$, we will have,

$$\mathbf{J}_l(k[\hat{\mathbf{x}}_i; \mathbf{v}_b]) = \mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]). \tag{3.23}$$

Note that $\mathbf{v}_b = \mathbf{x}_b$ is the case we discussed in Section 3.2.

With the above definitions, we can rewrite Eq.(3.18) as,

$$\mathscr{C}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b]) = \mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b])[\hat{\mathbf{x}}_i; \mathbf{v}_b]. \tag{3.24}$$

Together with the fact that $\mathbf{J}_l([\hat{\mathbf{x}}_i; \mathbf{v}_b])$ is piecewise constant, Eq.(3.24) reveals that $\mathscr{C}_l$ is a piecewise linear function which passes through the origin in $\mathbb{R}^{m_{-1}+M}$. Furthermore, Eq.(3.13), (3.15) can be rewritten as,

$$\mathbf{c}_l(\mathbf{x}_n) = \mathbf{J}_l(k[\hat{\mathbf{p}}_{-1}(\mathbf{x}_n); \mathbf{x}_b])[\hat{\mathbf{p}}_{-1}(\mathbf{x}_n); \mathbf{x}_b], 0 \leq l \leq L, k \in \mathbb{R}^+, \tag{3.25}$$

which proves the third equality in Eq.(3.2). For any piecewise linear activation function $\sigma(x)$, the following property holds,

$$\sigma(x_0) = x_0 \times \left.\frac{d\sigma(x)}{x}\right|_{x=x_0}, x_0 \in \mathbb{R}. \tag{3.26}$$

This implies that the proof works for any piecewise linear activation $\sigma(x)$ (whose derivative is piecewise constant) as long as we apply $k = 1$ and properly replace $\mathbf{\Sigma}_l$ in Eq.(3.21) according to the selected $\sigma(x)$.

3.5 Verification experiments

This section will further verify our theory experimentally.

As we mentioned in Section 3.3, there are two operational modes of CNN's bias:

1. Conventional parameters trained for adjusting the channels' output;

2. Auxiliary parameters trained for batch normalization.

We select three CNN models that involve either of these two operational modes. Also, these three models include both supported activation functions: ReLU and Leaky ReLU. Generally, we will verify the nontrivial Eq.(3.2) on every layer of the CNN (except the first layer, the reason has been discussed in Section 3.3). The hardware and software used for this verification are listed in Appendix B.1.

3.5.1 Pre-trained CNNs and their equivalent topologies

We elaborate training/validation/test settings on all models. We also discuss how we convert a trained CNN model to an equivalent version using Eq.(3.6).

3.5.1.1 Dataset and augmentation

We used the CIFAR-10 and CIFAR-100 ([46]; RGB images of $10/100$ classes; Resolution: 32×32; Pixel value range: $[0, 1]$; Each one includes two sets: $50k$ training, $10k$ test). In each dataset, training was conducted on $45k$ of the $50k$ set (randomly selected), and validation is conducted on the remaining $5k$; The test is performed on the $10k$ set. Data augmentation methods are employed in training. An input image sequentially goes through the random left or right flipping, the random adjustments of saturation(within $[0, 2.0]$)/contrast(within $[0.4, 1.6]$)/brightness(within 0.5), the random croppings to $32 \times 32 \times 3$ after resizing to $36 \times 36 \times 3$.

3.5.1.2 Models and their conversions

We use three standard models: VGG7 [11] with 7 ReLUs, ResNet20 ([12], code: [51]) with 20 Leaky ReLUs, and ResNet20-Fixup (the rightmost one in Figure 1 in [1], an input image requires normalization on its RGB channels using mean $[0.4914, 0.4822, 0.4465]$ and variance $[0.2023, 0.1994, 0.2010]$, code: [47]) with the Fixup initialization and 20 ReLUs. The first two use the second bias operational mode (batch normalization), and the last uses the first bias operational mode (conventional bias mode). Their parameters are listed in tables B.1, B.4, and B.5, respectively. VGG7 and ResNet20 are trained with BN layers first. After training, we extract $\mathbf{w}, \gamma, \beta, \mu, \sigma^2$ (Eq.(3.5)) to compute and collect the corresponding $\mathbf{w}'$ and $\mathbf{b}'$ with Eq.(3.6), layer by layer. Then, we rebuild every architectural layer with its $\mathbf{w}'$ and $\mathbf{b}'$ to construct an equivalent model (Fig.3.4). Similarly, ResNet20-Fixup is trained first; We extract all kernels and biases after training. We rebuilt every layer (we merge any convolutional layer having a multiplier, check the third column of table B.5) to construct an equivalent model. We verify that every rebuilt model achieves identical accuracy as its pre-trained version. We employ the rebuilt ones for our experiments.

3.5.1.3 Loss function, accuracy, training, validation, test

We used a summation of the kernels' regularization on the L_1 norm and the cross-entropy on a prediction's softmax as the loss function. We used Top-1 accuracy. Training, validation, and test were conducted on an RTX3090 GPU. All three CNN models were trained with Gradient Descent optimizers (GD) on a batch size of 100. Tables 3.1 and 3.2 showed additional details. We trained on the $45k$ samples every epoch and validated the trained model on the $5k$ samples every two epochs. The trained model would be saved if a higher validation accuracy was reached. We tested with the $10k$ test samples.

Models/lr intervals	1^{st} interval	2^{nd} interval	3^{rd} interval	Total	Test Acc
VGG7	$0.01, [0, 299]$	$0.002, [300, 399]$	$0.0005, [400, 500]$	501	89.5%
ResNet20	$0.01, [0, 99]$	$0.001, [100, 149]$	$0.0002, [150, 200]$	201	91.6%
ResNet20-Fixup	$0.002, [0, 79]$	$0.001, [80, 119]$	$0.0005, [120, 150]$	151	91.2%

Table 3.1: Details for training VGG7/ResNet20/ResNet20-Fixup on CIFAR-10. "$0.01, [0, 299]$" denotes the learning rate (lr) 0.01 during the training interval, maintained during the 0^{th} to the 299^{th} training epoch. Note that the classification accuracies are modest at best, but since our thrust is primarily theoretical, we did not attempt to further fine-tune the default hyperparameters in the available open-source implementations that we used.

Models/lr intervals	1^{st} interval	2^{nd} interval	3^{rd} interval	Total	Test Acc
VGG7	$0.01, [0, 499]$	$0.002, [500, 599]$	$0.0005, [600, 700]$	701	63.2%
ResNet20	$0.01, [0, 99]$	$0.001, [100, 149]$	$0.0002, [150, 200]$	201	68.8%
ResNet20-Fixup	$0.002, [0, 39]$	$0.001, [40, 79]$	$0.0005, [80, 120]$	121	64.0%

Table 3.2: Details for training VGG7/ResNet20/ResNet20-Fixup on CIFAR-100. Check Table 3.1 for notations.

3.5.2 Verifying Eq.(3.2) on the CNN models

As we mentioned, we will experimentally verify that $\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]) = \frac{[\mathbf{x}_n;\mathbf{x}_b]}{8}$ achieves the equality in Eq.(3.2).

We verify it on two types of layers: every Conv layer, and the FC layer. In other words, we verify that the linear activation value of an out-ch feature map's unit or the FC layer's output unit, computed through the original trained CNN, should be equal to the dot-product between the input $[\mathbf{x}_n; \mathbf{x}_b]$ and the reconstructed effective hyperplane $\mathbf{H}^{Adj}(\frac{[\mathbf{x}_n;\mathbf{x}_b]}{8})$. The verification can be described with two equations, Eq.(3.27) (Conv) and Eq.(3.28) (FC). Principles of the verification are illustrated using a symbol '$=$" in these previous figures: Fig.3.5(c) (Conv), and Fig.3.3(b) (FC).

$$c_{l,s,i} = \underbrace{\mathbf{F}_{l-1,s,i}([\mathbf{x}_n; \mathbf{x}_b]) \circledast \mathbf{w}_{l,s,i}}_{\text{Original CNN activation (Conv)}} = \hat{c}_{l,s,i} = \underbrace{\langle \mathbf{x}_n \mid \mathbf{H}^{Adj,I}_{l,s,i}(\frac{[\mathbf{x}_n; \mathbf{x}_b]}{8})\rangle + \langle \mathbf{x}_b \mid \mathbf{H}^{Adj,b}_{l,s,i}(\frac{[\mathbf{x}_n; \mathbf{x}_b]}{8})\rangle}_{\text{Reconstructed Conv unit activation using Adjoint}}, \tag{3.27}$$

$$c_{fc,k} = \underbrace{\mathbf{F}_{fc,k}([\mathbf{x}_n; \mathbf{x}_b])}_{\text{Original CNN activation (FC)}} = \hat{c}_{fc,k} = \underbrace{\langle \mathbf{x}_n \mid \mathbf{H}^{Adj,I}_{k}(\frac{[\mathbf{x}_n; \mathbf{x}_b]}{8})\rangle + \langle \mathbf{x}_b \mid \mathbf{H}^{Adj,b}_{k}(\frac{[\mathbf{x}_n; \mathbf{x}_b]}{8})\rangle}_{\text{Reconstructed FC unit activation using Adjoint}}, \tag{3.28}$$

where $c_{l,s,i}$ and $c_{fc,k}$ are true values (correspond to the left-hand side of the third "$=$" in Eq.(3.2)) while $\hat{c}_{l,s,i}$ and $\hat{c}_{fc,k}$ are approximated ones (correspond to the right-hand side of the third "$=$" in Eq.(3.2)); $c_{l,s,i}$ labels a unit at the stride move s, in the i^{th} out-ch feature map of the l^{th} Conv layer (i.e., a unit of $\mathbf{c}_l$); $c_{fc,k}$ labels the k^{th} entry in the $\mathbf{c}_{fc}$, from the FC layer, for activating a specific class unit. In detail, for an image $\mathbf{x}_n$, a dot-product should be verified to replicate a unit's linear activation value from the Conv or the FC layer, and this verification should go through units'/predicted classes' values of all layers except for the first one (which has been explained in Section 3.3); Also, $RM2$ is entangled with $RM4, RM3, RM1$ (Eq.(3.7)~(3.11)), which implies verifying $RM2$ is equivalent to verifying $RM4$~$RM1$. Since TensorFlow uses FP32 datatype (single precision) to compute a decimal, rounding errors are inevitable and result in a fractional mismatch between $\mathbf{c}_l$ and $\hat{\mathbf{c}}_l$ due to different computation paths. We measure relative errors ϵ for evaluating the approximations, i.e.,

$$\epsilon_l = \frac{\hat{\mathbf{c}}_l - \mathbf{c}_l}{\mathbf{c}_{l,\neq 0}}, \tag{3.29}$$

where l is either the Conv layer index or the FC layer; $\mathbf{c}_{l,\neq 0}$ substitutes all zeros inside $\mathbf{c}_l$ with the smallest positive float of FP32 to avoid any divide by zero exception.

3.5.3 Results

We verify our CNN's unit reconstruction approach on VGG7/ResNet20/ResNet20-Fixup with CIFAR-10 and CIFAR-100 test sets. Relative errors ϵ_l are collected layer by layer over the $10k$ test samples for a CNN model. All relative errors are shown as histograms in Figures 3.6, 3.7, 3.8, 3.9, 3.10, and 3.11. VGG7 has 6 histograms (5 convs + 1 FC), and ResNet20/ResNet20-Fixup each have 19 histograms (18 convs + 1 FC). Percentages of units with reconstruction error $\epsilon_l \leq 1\%$ collected from all models' layers are listed in Tables 3.3 and 3.4. In summary, all three models show that over 99.97% of the units have relative reconstruction errors $\leq 1\%$. These results experimentally validate that $\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b]) = \frac{[\mathbf{x}_n; \mathbf{x}_b]}{8}$ achieves Eq.(3.2).

Therefore, the soundness of our theory is experimentally confirmed.

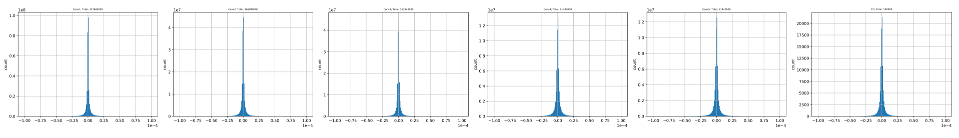

(a_1) (a_2) (a_3) (a_4) (a_5) (a_6)

Figure 3.6: Histograms of relative errors (Eq.(3.29)) between units directly computed by CN-N ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-10 using VGG7. The x-axis and y-axis indicate the error and frequency ($10k$ test samples), respectively. a_1~a_5 is collected from Conv1~5 layer, and a_6 is collected from the FC layer. In terms of the 4^{th} column of table.B.1, a_1 to a_6 should have relative errors from $327.68m (= 32,768 \times 10k), 163.84m (= 16,384 \times 10k), 163.84m, 61.44m (= 6,144 \times 10k), 61.44m$, and $100k$, respectively. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.3.

ϵ_l/ % / Models	VGG7	ResNet20	ResNet20-Fixup
ϵ_1	99.9987	99.9999	99.9972
ϵ_2	99.9978	99.9994	99.9992
ϵ_3	99.9985	99.999	99.9993
ϵ_4	99.9935	99.9977	99.9989
ϵ_5	99.9933	99.9974	99.9974
ϵ_6	99.991 (FC)	99.9919	99.9975
ϵ_7	N/A	99.9835	99.9914
ϵ_8	N/A	99.9839	99.9946
ϵ_9	N/A	99.986	99.9954
ϵ_{10}	N/A	99.9877	99.9959
ϵ_{11}	N/A	99.9835	99.9953
ϵ_{12}	N/A	99.9871	99.9955
ϵ_{13}	N/A	99.9807	99.9935
ϵ_{14}	N/A	99.9808	99.9935
ϵ_{15}	N/A	99.978	99.9943
ϵ_{16}	N/A	99.9781	99.9934
ϵ_{17}	N/A	99.9764	99.9929
ϵ_{18}	N/A	99.978	99.9927
ϵ_{19}	N/A	99.989 (FC)	99.992 (FC)

Table 3.3: Percentages (%) of units with reconstruction error $\epsilon_l \leq 1\%$ collected from all layers (involved in verification experiments) of VGG7/ResNet20/ResNet20-Fixup over the CIFAR-10 test set ($10k$ samples). N/A indicates that the layer does not exist (VGG7 has only 6 layers, and its Conv0 is not valid in our analysis scope).

ϵ_l/ % / Models	VGG7	ResNet20	ResNet20-Fixup
ϵ_1	99.9988	99.9997	99.9997
ϵ_2	99.9978	99.9995	99.9995
ϵ_3	99.998	99.9988	99.9993
ϵ_4	99.9928	99.9963	99.9992
ϵ_5	99.993	99.9968	99.9989
ϵ_6	99.9807 (FC)	99.9833	99.9967
ϵ_7	N/A	99.9789	99.9896
ϵ_8	N/A	99.9751	99.9951
ϵ_9	N/A	99.9878	99.9952
ϵ_{10}	N/A	99.9808	99.9953
ϵ_{11}	N/A	99.979	99.994
ϵ_{12}	N/A	99.9839	99.9953
ϵ_{13}	N/A	99.9767	99.9921
ϵ_{14}	N/A	99.9746	99.9935
ϵ_{15}	N/A	99.9744	99.9941
ϵ_{16}	N/A	99.9726	99.993
ϵ_{17}	N/A	99.9705	99.9935
ϵ_{18}	N/A	99.9705	99.9949
ϵ_{19}	N/A	99.9777 (FC)	99.9896 (FC)

Table 3.4: Percentages (%) of units with reconstruction error $\epsilon_l \leq 1\%$ collected from all layers of VGG7/ResNet20/ResNet20-Fixup over the CIFAR-100 test set ($10k$ samples). Check Table 3.3 for details.

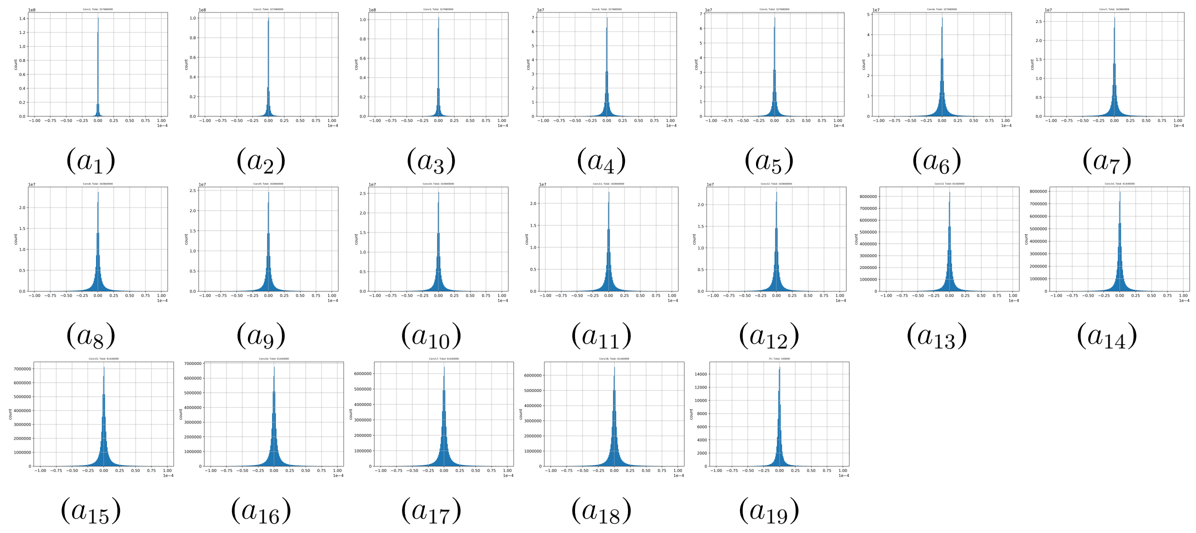


Figure 3.7: Histograms of relative errors (Eq.(3.29)) between units directly computed by CNN ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-10 using ResNet20. Similar to Fig.3.6, a_1~a_{18} is collected from Conv1~18 layer, and a_{19} is collected from the FC layer. In terms of the 5^{th} column of table.B.4, a_1 to a_{18} should have relative errors from $327.68m(a_1$~$a_6)$, $163.84m(a_7$~$a_{12})$, $61.44m(a_{13}$~$a_{18})$, and $100k(a_{19})$, respectively. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.3.

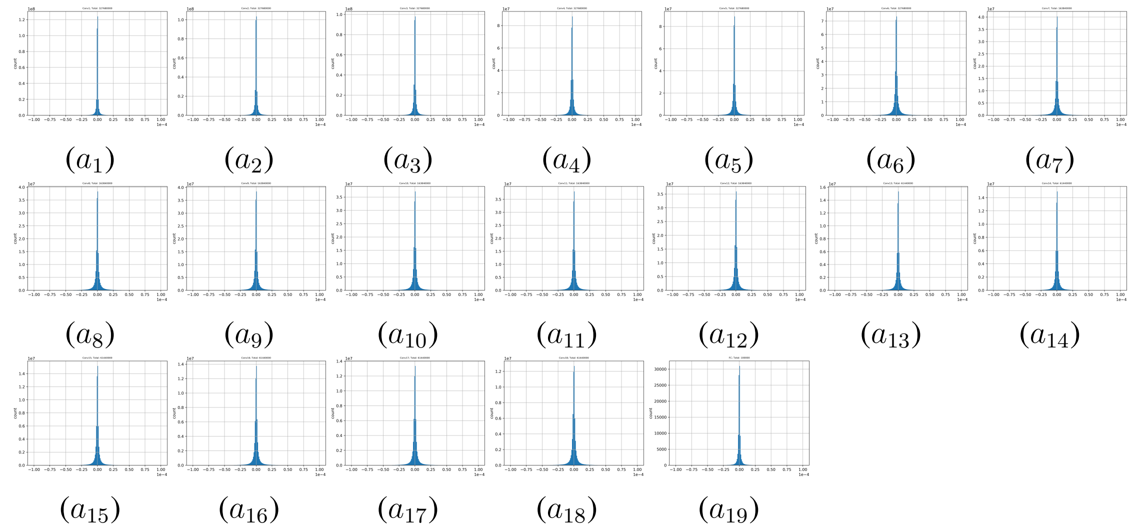


Figure 3.8: Histograms of relative errors (Eq.(3.29)) between units directly computed by CNN ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-10 using ResNet20-Fixup. Details are similar to Fig.3.7. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.3.

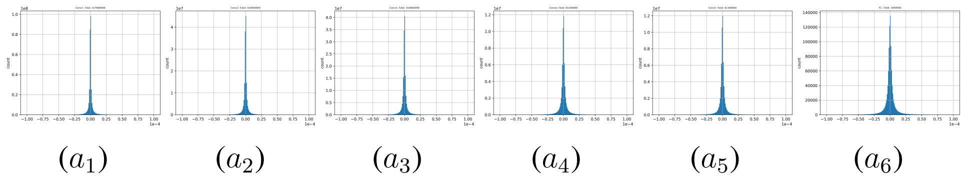


Figure 3.9: Histograms of relative errors (Eq.(3.29)) between units directly computed by CNN ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-100 using VGG7. Details are similar to Fig.3.6 except that the quantity of relative errors in a_6 is $1m$. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.4.

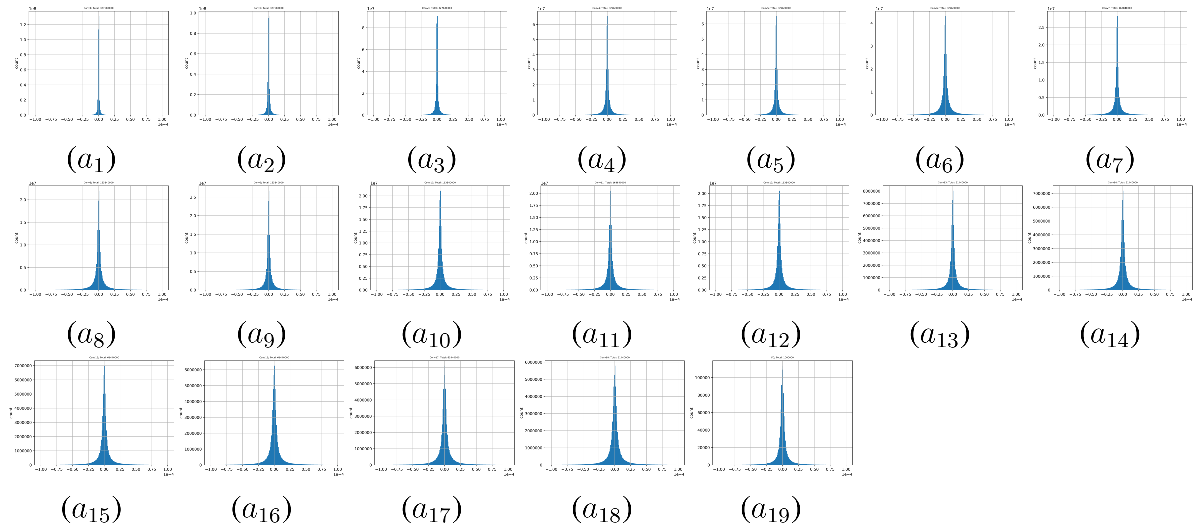


Figure 3.10: Histograms of relative errors (Eq.(3.29)) between units directly computed by CNN ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-100 using ResNet20. Details are similar to Fig.3.7 except that the quantity of relative errors in a_{19} is $1m$. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.4.

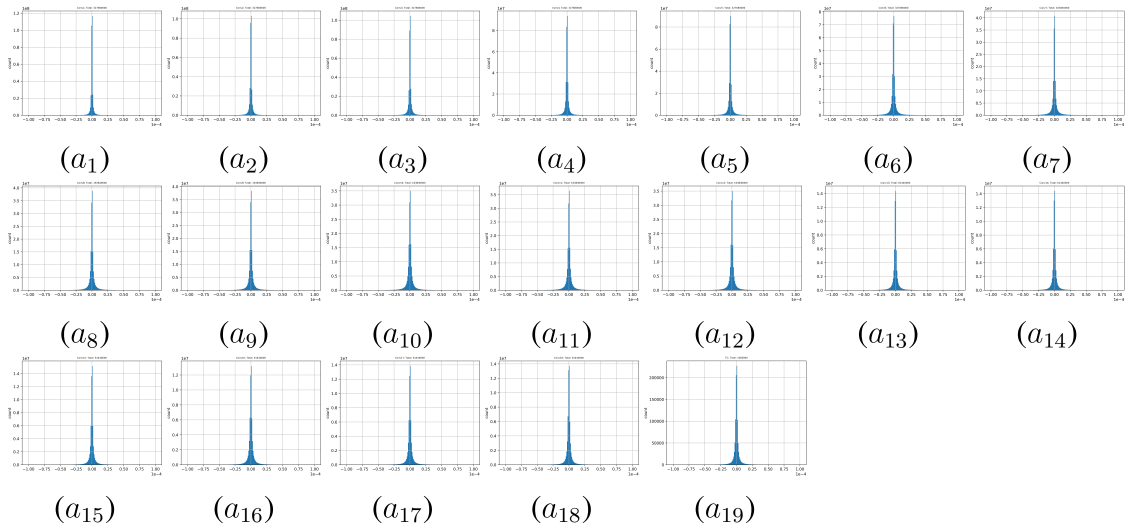


Figure 3.11: Histograms of relative errors (Eq.(3.29)) between units directly computed by CNN ($\mathbf{c}_l$) and their values reconstructed by our method ($\hat{\mathbf{c}}_l$) on CIFAR-100 using ResNet20-Fixup. Details are similar to Fig.3.7 except that the quantity of relative errors in a_{19} is $1m$. Percentages of $\epsilon_l \leq 1\%$ for all subplots are listed in Table 3.4.

4. CNN'S BEHAVIORS UNDER ADVERSARIAL ATTACKS

4.1 Overview

We present our theoretical framework in the analysis of adversarial attacks. We propose three top-down experiments that rip through CNN's internal response to these attacks. We find that CNN's decision process is extremely sensitive to adversarial experiments. This sensitive property results in large fluctuations in the CNN's decision when facing two visually identical images. The sensitivity actually results from a weakness inside the CNN – its decision process lacks the continuity that our human vision has.

4.2 Applications: Analyzing adversarial attacks to CNN without bias

This section studies CNN models (without bias) under adversarial attacks with our Algorithm 1. Generally, we propose three experiments to explore CNN's decision process gradually. We first analyze all the FC units' effective hypersurfaces under one type of adversarial attacks and then select one of these effective hypersurfaces as the representative to further probe its property under various adversarial attacks. In detail, we select a "horse" image $\mathbf{x}_0$ (its label index is 7) from the CIFAR-10 data set as the original image; Experiment 1, using $RM0$, visualizes the variations of 10 hypersurfaces from each of the 10 outputs of a trained model under adversarial noise; Experiment 2 observes a single effective hypersurface and explores its variations under different targeted adversarial noise; Experiment 3 further probes the effective hypersurface's variations through comparisons between scaled-down adversarial noise (still misclassified) and Gaussian noise (correctly classified) generated under the same noise conditions (mean and variance). We use $\mathbf{Advr}$ to denote an adversarial "noise" and keep using $\mathbf{z}(\mathbf{x}) = \frac{\mathbf{x}}{8}$.

4.2.1 Experiment 1: Hypersurfaces reconstructed with or without an adversarial noise

We use the "basic iterative method" from [2, 3] to compute an adversarial noise. The factor in [2] is 0.007. We use a factor of 0.04 (around $5\times$) because we normalize an input image with the stds around 0.2. Computed "noise" is added to the input image, and we verify that the adversarial input fools VGG7 to predict “dog”, based on the “horse” input. We repeat the same experiment for Fixup-ResNet20. We first have to verify Eq.(2.17) before visualizing the hyperplanes. The output reconstruction results (Tables 4.1 and 4.2) show the output of the CNN in response to the adversarial input $\mathbf{x}_0 + \mathbf{Advr}$ (column marked “CNN”) and the activities based on the reconstructed hyperplanes (columns marked “Recon w/ $\mathbf{H}_{adv}$” and “Recon w/ $\mathbf{H}_{orig}$”). See the table caption for details. The results show that with the hyperplane reconstructed using the matching input (in this case, the adversarial input $\mathbf{x}_0 + \mathbf{Advr}$), precise output values can be replicated (“Recon w/ $\mathbf{H}_{adv}$” column), while the values are off when non-matching input is used to reconstruct the hyperplane (“Recon w/ $\mathbf{H}_{orig}$” column). This effect can be quite severe, as shown in the case of the Fixup-ResNet20 (Table 4.2), since even the reconstructed output class is different (“cat”) from that of the CNN (“dog”).

4.2.2 Experiment 2: Measuring the spread of a hypersurface in response to targeted adversarial noise

We use the “iterative least-likely class method” from [3] to generate 9 targeted adversarial noise patterns for the horse image, $S_a = \{\mathbf{Advr}_i \mid i \in \{0, 1, .., 9\}\}$ where $\mathbf{Advr}_7 = \theta_{\mathcal{X}}$ (no noise: 7 is the “horse” image’s label index). With such adversarial noise, our models (VGG7 and Fixup-ResNet20) can be fooled to produce any of the 9 incorrect classes. We first visualize the reconstructed hyperplanes from the unit $k = 7$ (the “horse” unit) under targeted adversarial attack (Fig.4.4(a_1) and (a_2)). To see how these hyperplanes are spread out, we used tSNE [52] to project these hyperplanes in 3D space, and randomly select 10 more images ($\{\mathbf{x}_1, ..., \mathbf{x}_{10}\}$) of different classes (not “horse”) for comparison. Specifically, we project $\{\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_i)) \mid i \in \{0, 1, ..., 9\}\} \cup \{\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_j)) \mid j \in \{1, 2, ..., 10\}\}$ using tSNE for analysis. We illustrate the procedures in Fig.4.1.

Class index k / Method	CNN	Recon w/ $\mathbf{H}_{adv}$	Recon w/ $\mathbf{H}_{orig}$
0 (airplane)	8.31594	8.315954	8.6690855
1 (automobile)	4.0766134	4.0766125	4.471591
2 (bird)	7.1327333	7.132732	7.3968844
3 (cat)	8.730146	8.730144	8.764506
4 (deer)	7.252265	7.252265	7.06274
5 (dog)	**9.732837**	**9.732836**	**9.646278**
6 (frog)	5.3150105	5.315008	5.101524
7 (horse)	9.613704	9.613699	7.970075
8 (ship)	-1.3195618	-1.3195606	-1.1485753
9 (truck)	6.7904973	6.7904935	7.129147

Table 4.1: Experiment 1, VGG7 (w/o bias): Verifying the reconstructed output (Eq.(2.17)) when adversarial noise is added. The original image $\mathbf{x}_0$ was a "horse" class. However, after adding adversarial noise $(\mathbf{x}_0+\mathbf{Advr})$, the CNN predicts "dog" (marked in bold). The second column (CNN) shows the CNN's output layer activities given the adversarial input $(\mathbf{x}_0+\mathbf{Advr})$. The third column (Recon w/ $\mathbf{H}_{adv}$) shows the reconstructed output based on the hyperplane $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0+\mathbf{Advr}))$ reconstructed from the same adversarial input. We can see that the output values are virtually identical to those of the CNN in column 2. The fourth column (Recon w/ $\mathbf{H}_{orig}$) shows the reconstructed output based on the hyperplane $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))$, this time, reconstructed from the original (non-adversarial) input $\mathbf{x}_0$. That is, in this case, $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))$ was multiplied by the adversarial input $(\mathbf{x}_0+\mathbf{Advr})$. We see that the slightly different reconstructed hyperplane (with or without adversarial noise) leads to a slight difference in the final output value. This shows that our AdjointBackMap gives precise replication of the CNN's original output values, and this depends on the use of the original input in the computation of the reconstructed hyperplane.

Class index k / Method	CNN	Recon w/ $\mathbf{H}_{adv}$	Recon w/ $\mathbf{H}_{orig}$
0 (airplane)	-0.9798855	-0.97988594	0.0621146
1 (automobile)	-2.5764616	-2.5764558	-0.5108745
2 (bird)	-0.5919545	-0.59196144	-6.3072925
3 (cat)	10.356275	10.356285	**7.726749**
4 (deer)	5.80124	5.801249	-8.883365
5 (dog)	**10.47916**	**10.47917**	-0.43725738
6 (frog)	3.5252628	3.5252705	-1.7145596
7 (horse)	6.5016255	6.5016294	-19.942373
8 (ship)	-6.2591143	-6.2591157	-1.7208233
9 (truck)	2.1569839	2.1569812	-0.61982846

Table 4.2: Experiment 2, Fixup-ResNet20 (w/o bias): Verifying the reconstructed output (Eq.(2.17)) when adversarial noise is added.

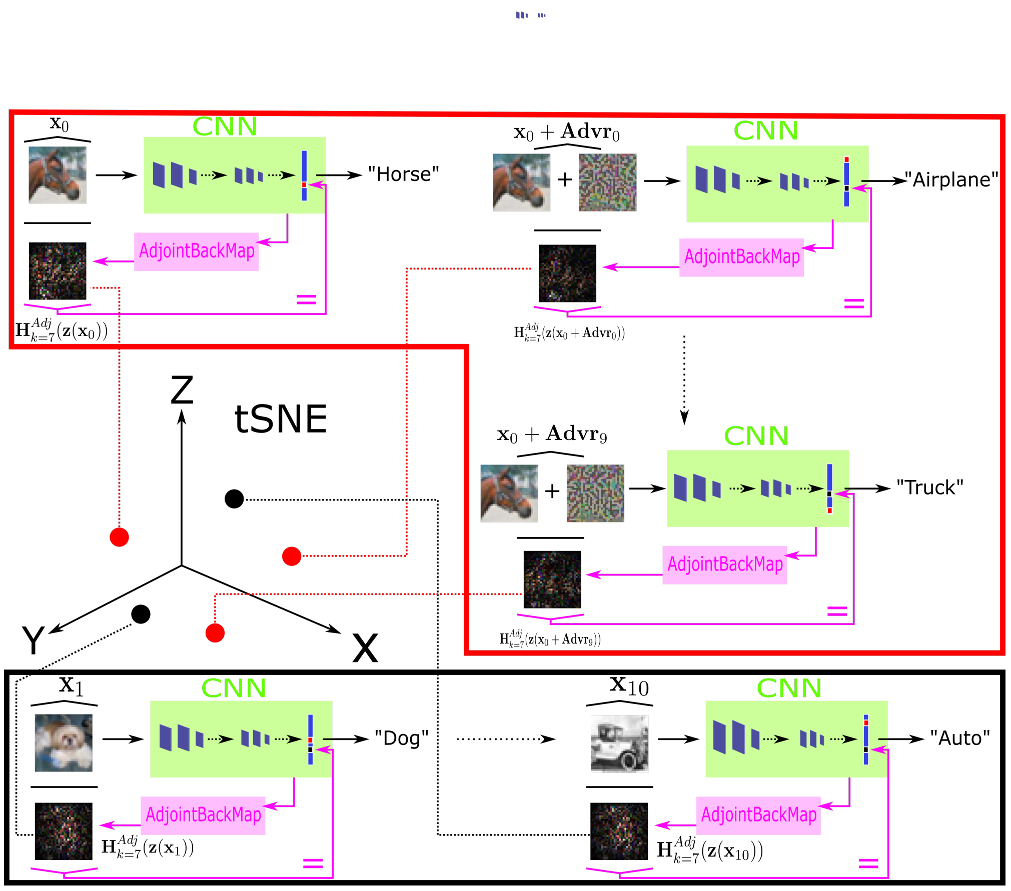

$\mathbf{x}_0$
CNN
"Horse"
AdjointBackMap
=
$\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0))$
$\mathbf{x}_0 + \mathbf{Advr}_0$
+
CNN
"Airplane"
AdjointBackMap
=
$\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_0))$
Z
tSNE
Y
X
$\mathbf{x}_0 + \mathbf{Advr}_9$
+
CNN
"Truck"
AdjointBackMap
=
$\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_9))$
$\mathbf{x}_1$
CNN
"Dog"
AdjointBackMap
$\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_1))$
=
$\mathbf{x}_{10}$
CNN
"Auto"
AdjointBackMap
$\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_{10}))$
=

Figure 4.1: Procedures for Experiment 2 (for models w/o bias). In general, this experiment is designed to numerically measure the fluctuation of a single effective hypersurface in response to different adversarial noise. Red box: We first generate 9 targeted adversarial noise, $\{\mathbf{Advr}_i \mid i \in \{0, 1, ..., 9\}\}$, which can fool a CNN model to incorrectly classify the "horse" ($\mathbf{x}_0$, its label index $= 7$) to the 9 classes, where $\mathbf{Advr}_7 = \theta_{\mathcal{X}}$ denotes no noise. Black box: Then, we collect the 10 effective hyperplanes in response to these adversarial noise, $\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x_0} + \mathbf{Advr_i}))$. XYZ axes: At last, we use tSNE to project these hyperplanes (red) in 3D space with 10 more hyperplanes (black) of images having different classes ($\{\mathbf{x}_i \mid i \in \{1, 2, ..., 10\}\}$, not "horse") for comparison. Note adversarial attacks do not affect our reconstruction capability (marked by "=" in the figure. We rotate the dot-product "$\langle \cdot \mid \cdot \rangle$" to "$\overbrace{\underbrace{\frac{\cdot}{\cdot}}}$" for a compact layout.). Check Section 4.2 for details.

4.2.3 Experiment 3: Comparing the spread of hypersurfaces in response to adversarial vs. Gaussian noise

First, we prepare 50 adversarial noise patterns S_b, based on S_a in Experiment 2, which are the scaled-down version of those in S_a. That is, $\beta \times \mathbf{Advr}_i$, where β is a small constant between 0.0 and 1.0. We make sure that images with added noise from S_b all lead to incorrect classification by the CNN (VGG7 or Fixup-ResNet20).

Next, we generate 50 Gaussian noise patterns ($32 \times 32 \times 3$) using the same pixel mean and variance statistically computed from S_b. Let us call these Gaussian-noise patterns S_g. When added to the original image (the horse image), none of the Gaussian noise patterns in S_g can fool our VGG7 or Fixup-ResNet20.

We project the noise-added input-based hyperplanes for output unit $k = 7$, $\{\mathbf{H}_{k=7}^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{n})) \mid \mathbf{n} \in S_b \bigcup S_g\}$, using Factor Analysis [52]. We repeat the experiment for both VGG7 and Fixup-ResNet20. The results are shown in Fig.4.5.

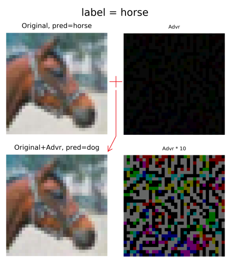
4.2.4 Results

Experiment 1 is illustrated in Figures 4.2 and 4.3; Experiments 2 and 3 are illustrated in Figures 4.4 and 4.5, respectively. See figures' captions for details.

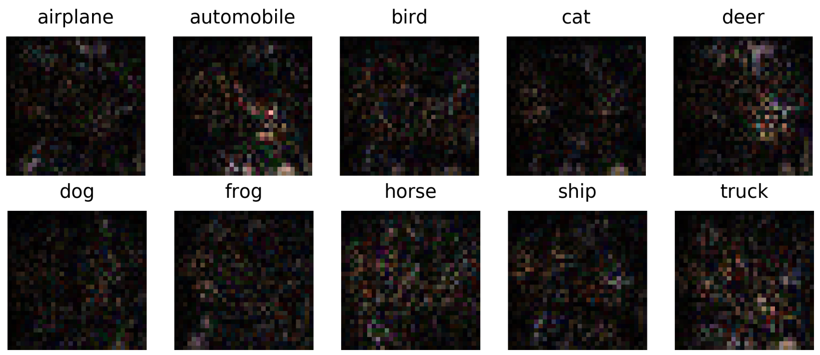
4.2.5 Analysis

Experiment 1 reveals that $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}))$ (with noise) is significantly different from $\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))$ (without noise) for all classes (differences are directly illustrated in Fig.4.2(c) and 4.3(c). We learn from their a_5 that this difference starts from Conv1, the second conv layer, through the $RM3$ reconstruction on that layer, although no kernel in that layer changes. It implies that effective hypersurfaces are very sensitive in response to small changes in the input, and either VGG7 or Fixup-ResNet20 takes different roads to determine the final output of two perceptually identical images, which is different from humans who may be tolerant to such small variations in pixel value.

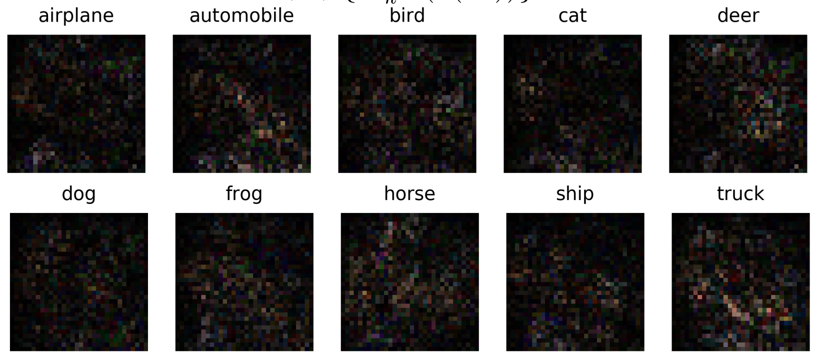
Experiment 2 (Fig.4.4) shows that although the adversarial noise has a very low magnitude,



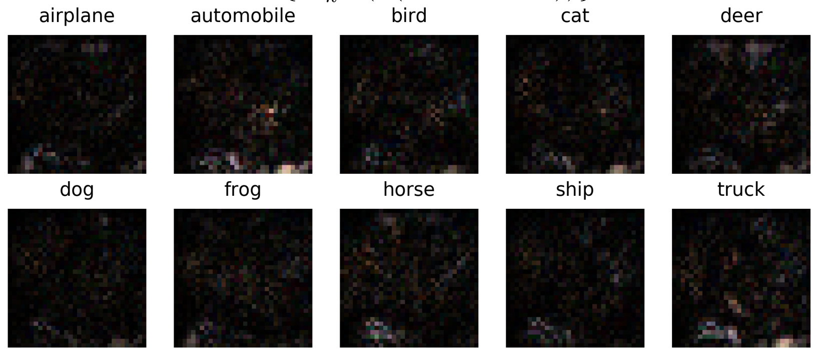
(a_1) $\mathbf{x}_0 + \mathbf{Advr}$



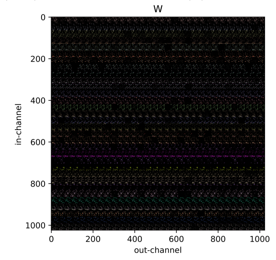
(a_2) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}$



(a_3) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}))\}$

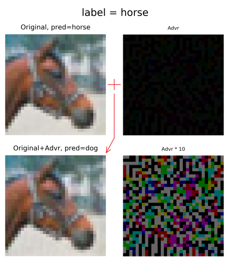


(a_4) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr})) - \mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}$

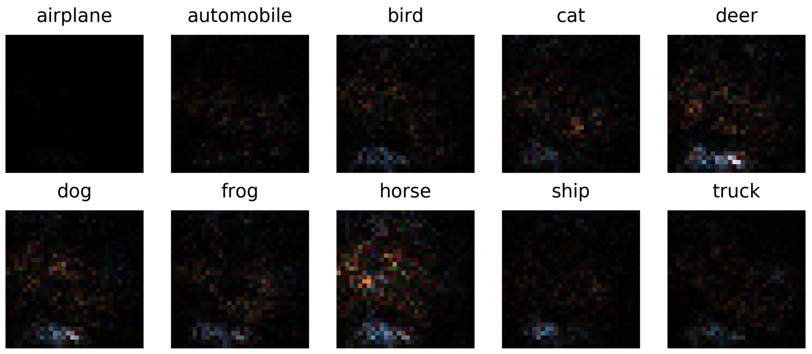


(a_5) $\mathbf{H}_{\{j,i\}}^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr})) - \mathbf{H}_{\{j,i\}}^{Adj}(\mathbf{z}(\mathbf{x}_0))$

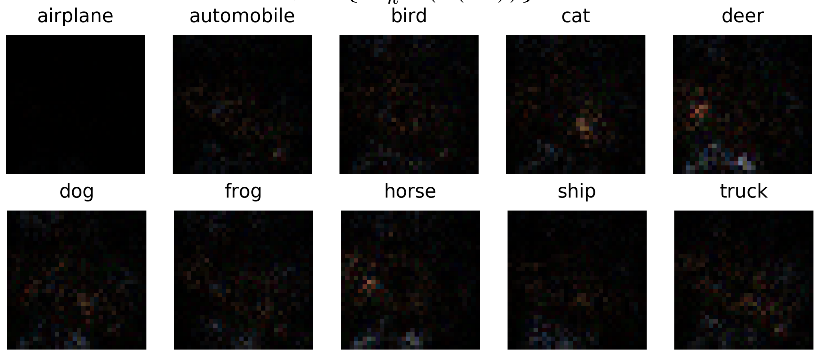
Figure 4.2: Experiment 1, VGG7 (w/o bias): Visualizing VGG7's hyperplanes under an adversarial example ("horse" to "dog"). (a_1) Original "Horse" (top left), adversarial noise (top right), and the final adversarial input (bottom left). Effective hyperplanes of the FC's prediction ($RM0$) under the original input (a_2), under the adversarial input (a_3), and the differences between a_2 and a_3 (a_4). (a_5) Differences between original and adversarial-based hyperplanes using $RM3$ on Conv1 (low-level units, and $\mathbf{H}_{\{j,i\}}^{Adj}$ refers to eq.(21) in Appendix of [5]).



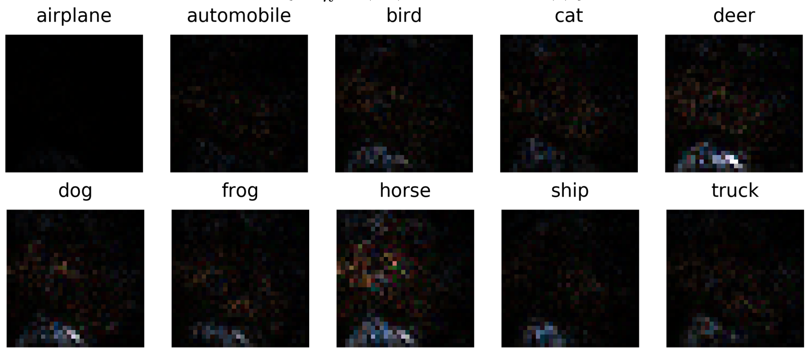
(a_1) $\mathbf{x}_0 + \mathbf{Advr}$



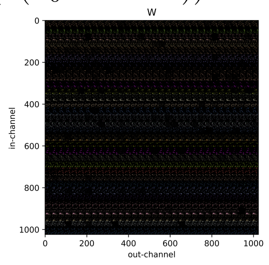
(a_2) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}$



(a_3) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}))\}$

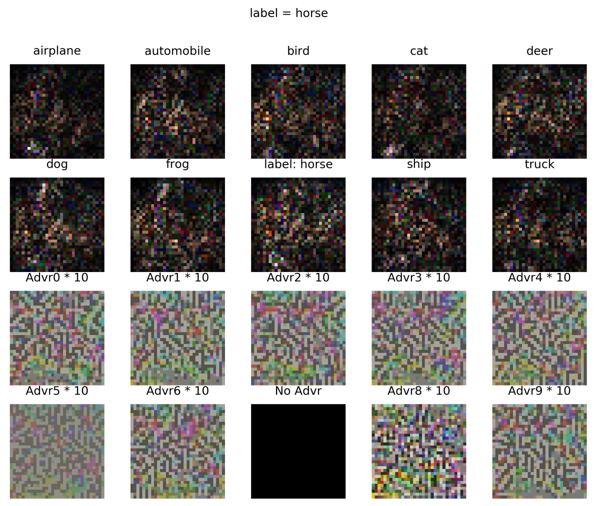


(a_4) $\{\mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr})) - \mathbf{H}_k^{Adj}(\mathbf{z}(\mathbf{x}_0))\}$



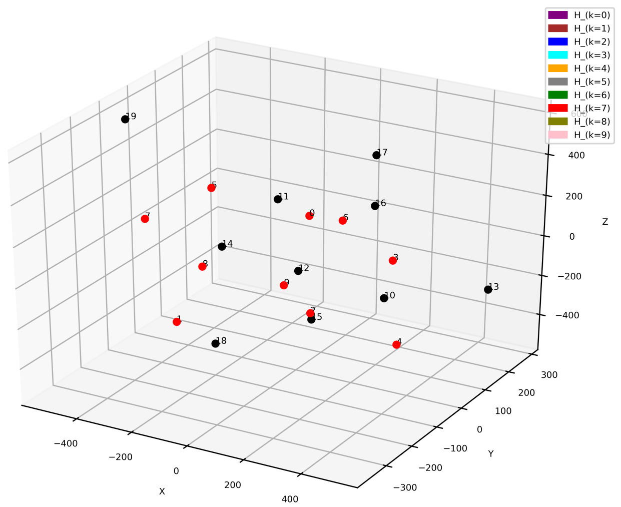
(a_5) $\mathbf{H}_{\{j,i\}}^{Adj}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr})) - \mathbf{H}_{\{j,i\}}^{Adj}(\mathbf{z}(\mathbf{x}_0))$

Figure 4.3: Experiment 1, Fixup-ResNet20 (w/o bias): Visualizing Fixup-ResNet20's hyperplanes under adversarial input ("horse" to "dog"). The format is the same as Fig.4.2. (a_5) is mapped from Conv1 of Residual Block 0.



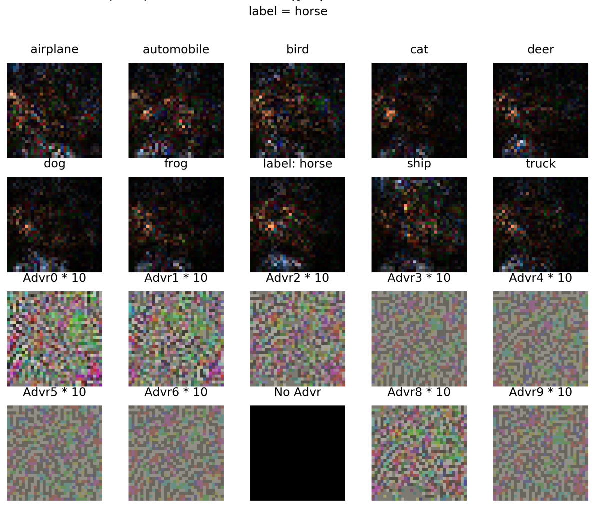
VGG7

(a_1) $\mathbf{Advr}_i$ and $\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_i))$



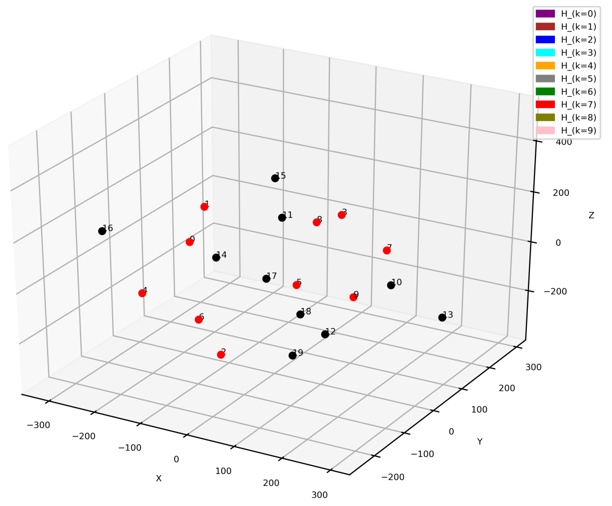
VGG7

(b_1) tSNE proj: adv vs. different classes



Fixup-ResNet20

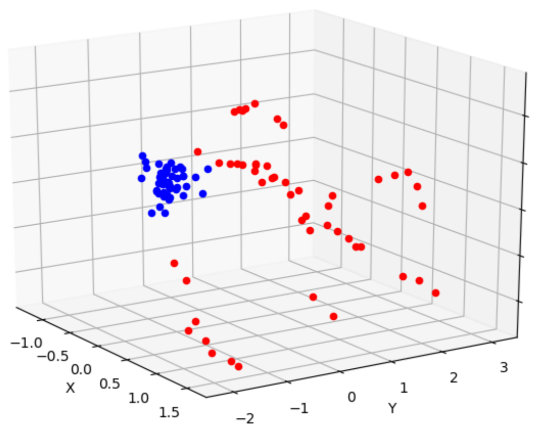
(a_2) $\mathbf{Advr}_i$ and $\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_i))$



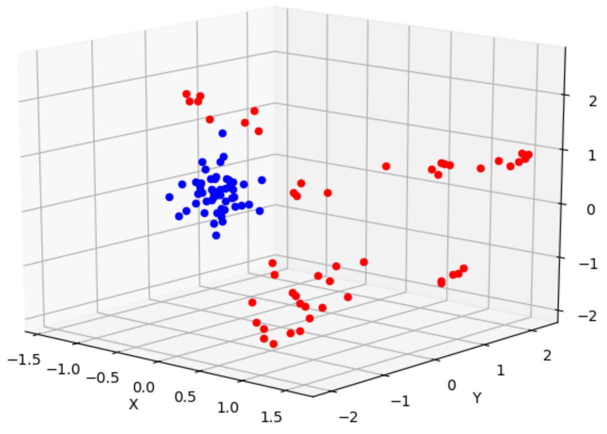
Fixup-ResNet20

(b_2) tSNE proj: adv vs. different classes

Figure 4.4: Experiment 2 (for models w/o bias): Effective hypersurfaces of unit $k = 7$ (horse unit) under targeted attacks and their tSNE projections. Top row: VGG7; Bottom row: Fixup-ResNet20. For VGG7, (a_1) Hyperplanes for the same output unit $k = 7$ (correct class index) under targeted adversarial noise S_a: $\{\mathbf{H}^{Adj}_{k=7}(\mathbf{z}(\mathbf{x}_0 + \mathbf{Advr}_i)) \mid \mathbf{Advr}_i \in S_a\}$ by $RM0$ (see text for details). The top two rows show the hyperplanes of unit $k = 7$, reconstructed from different adversarial inputs (based on the same horse image). The two bottom rows are the corresponding targeted adversarial noise ($10\times$, for visualization purposes; the black one means 0 adversarial noise, for the correct class "horse"). (b_1) tSNE projections of the hyperplanes in a_1 (red points), plotted along with the projections of hyperplanes for unit $k = 7$ given images of different classes, i.e., non-horse classes (black points). Note that the spread due to adversarial noise and spread due to class variation are almost the same. (a_2) and (b_2) show the same as above for Fixup-ResNet20. The same kind of spread can be seen.



(a) VGG7



(b) Fixup-ResNet20

Figure 4.5: Experiment 3 (for models w/o bias): Factor Analysis of the effective hypersurface with down-scaled adversarial noise compared to Gaussian noise. (**Red**) 50 effective hyperplanes computed from 50 adversarial examples (original image plus adversarial noise from S_b); (**Blue**) 50 effective hyperplanes computed from 50 non-adversarial examples (original image plus Gaussian noise from S_g). Effective hyperplanes of adversarial cases (Red) are significantly more spread out than non-adversarial cases (Blue).

the resulting distances between reconstructed hyperplanes are very large (red points in the figure). We can appreciate more how large the gaps are between these hyperplanes when comparing their spread to those of the reconstructed hyperplanes from input images of a totally different class (black points in the figure).

Further, Experiment 3 uses scaled-down adversarial noise from Experiment 2 to lower the magnitude of adversarial noise while maintaining their adversarial property (i.e., misclassification). The spread of the corresponding reconstructed hyperplanes is still great. The spread is significantly greater than the reconstructed hyperplanes based on Gaussian-noise-added inputs with the same noise mean and variance. Fig.4.5 shows the stark difference in how these reconstructed hyperplanes behave (red: image + adversarial noise; blue: image + Gaussian noise).

These experiments reveal that an effective hypersurface is brittle to adversarial noise despite being visually indistinguishable to the human eyes. Visually similar images can easily knock off a CNN by misleading its decision process because CNN is essentially different from our human vision. Our AdjointBackMap-based analysis allows us to gain insights on why this is the case, through experiments like those carried out above.

4.3 Applications: Analyzing adversarial attacks to CNN with bias

This section will generalize the sensitivity analysis to models with bias considered. We will revise the three experiments inherited from Section 4.2 to analyze effective hypersurfaces for CNN using bias.

We use the same three models, VGG7, ResNet20, and ResNet20-Fixup, and the same setting of $\mathbf{z}([\mathbf{x}_n; \mathbf{x}_b])$ as Section 3.5 for our experiments. The dataset is the same as Section 4.2. We use $\mathbf{Advr}$ to denote the adversarial "noise".

4.3.1 Experiment 1: Visualize effective hypersurfaces with and without adversarial noise

We will visualize the effective hypersurfaces of $RM0$ and observe their differences with and without the adversarial noise. Given a typical input image $\mathbf{x}_0$ ("Bird"), we employ the "basic iterative method" (an untargeted attack method from [3]) to generate an adversarial noise sample. Such a sample can deceive a CNN model when adding to the image. Unlike Experiment 1 in Section 4.2, the effective hypersurface $\mathbf{H}_k^{Adj}$ of a usual CNN model is composed of two parts: $\mathbf{H}_k^{Adj,I}$ and $\mathbf{H}_k^{Adj,b}$. We will normalize $\{\mathbf{H}_k^{Adj,I}\}_{k=0}^{9}$ to the same scale for visualization. For VGG7 or ResNet20, we will reshape its $\mathbf{H}_k^{Adj,b}$ to a square before visualization; For ResNet20-Fixup, the reshape will not be applied since its $\mathbf{H}_k^{Adj,b}$ is small-dimension (dimensions are listed in table B.2 or B.3, Appendix). Before the experiment, we verified that the CNN's decision hyperplanes for adversarial example $(\mathbf{x}_0 + \mathbf{Advr})$ are $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}; \mathbf{x}_b]))\}_{k=0}^{9}$, and incorrect input might result in low precision of reconstruction (Check Tables 4.3, 4.4, 4.5). We illustrate visualization results in Fig.4.6, 4.7, 4.8.

4.3.2 Experiment 2: Visualize an effective hypersurface in response to different adversarial noise

We will focus on one of the effective hypersurfaces and explore its variations under different adversarial noise. We select the effective hypersurface for the label ("Bird") unit, $\mathbf{H}_{k=2}^{Adj}$. Given a CNN model, we generate 9 adversarial noise samples using the "iterative least-likely class method" from

Class (k) / Method	VGG7's FC	Recon w/ $\mathbf{H}_{adv}$	Recon w/ $\mathbf{H}_{orig}$
airplane (0)	4.654	4.655	-4.927
automobile (1)	-0.194	-0.194	3.109
bird (2)	10.561	10.561	-22.695
cat (3)	**14.794**	**14.794**	8.459
deer (4)	7.060	7.060	6.703
dog (5)	8.961	8.961	**17.681**
frog (6)	0.329	0.329	-3.711
horse (7)	2.009	2.009	-2.211
ship (8)	1.153	1.153	3.265
truck (9)	3.127	3.127	6.592

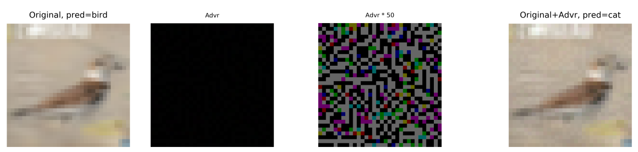
Table 4.3: Experiment 1, VGG7 (with bias): Verifying the effective hyperplanes' reconstruction for an adversarial example. $\mathbf{H}_{orig}$ denotes the effective hyperplane for the original "Bird" image, i.e., $\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0; \mathbf{x}_b]))$. $\mathbf{H}_{adv}$ denotes the effective hyperplane for the adversarial example, i.e., $\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}; \mathbf{x}_b]))$, where Fig.4.6 illustrates $\mathbf{Advr}$. VGG7 predicts the second column's values. The third or fourth column lists dot-products (reconstructed values) between the adversarial example and the effective hyperplane of the adversarial example or the original "Bird" image, respectively. The fourth column deviates from the predicted values and even hits an incorrect class. This implies that a correct input is essential for maintaining reconstruction precision.

Class (k) / Method	VGG7's FC	Recon w/ $\mathbf{H}_{adv}$	Recon w/ $\mathbf{H}_{orig}$
airplane (0)	-0.575	-0.575	1.683
automobile (1)	-3.718	-3.718	14.552
bird (2)	3.780	3.780	-33.297
cat (3)	7.480	7.480	20.085
deer (4)	**9.631**	**9.631**	**20.219**
dog (5)	2.895	2.895	20.013
frog (6)	-4.851	-4.851	-18.750
horse (7)	-4.921	-4.921	6.888
ship (8)	1.068	1.068	7.762
truck (9)	-0.191	-0.191	8.563

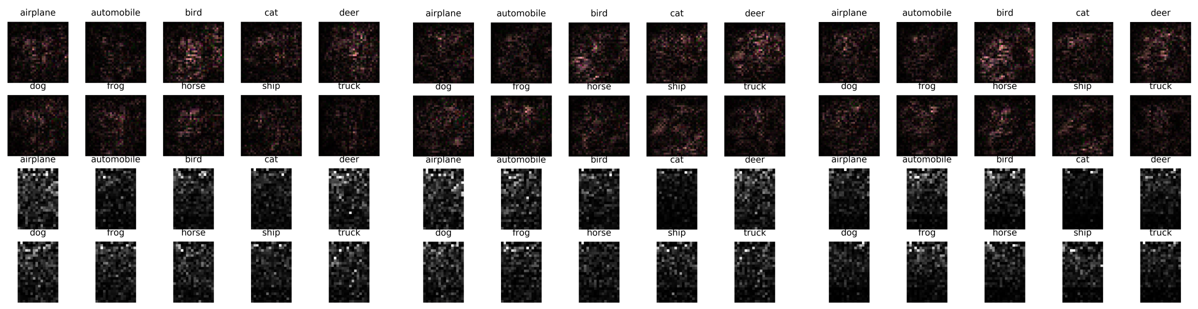
Table 4.4: Experiment 1, ResNet20 (with bias): Verifying the effective hyperplanes' reconstruction for an adversarial example. Fig.4.7 illustrates the adversarial noise $\mathbf{Advr}$. Check Table 4.3 for details.

Class (k) / Method	VGG7's FC	Recon w/ $\mathbf{H}_{adv}$	Recon w/ $\mathbf{H}_{orig}$
airplane (0)	**8.417**	**8.417**	3.307
automobile (1)	2.965	2.965	3.379
bird (2)	4.406	4.406	-53.405
cat (3)	4.508	4.508	-20.650
deer (4)	7.583	7.583	-12.317
dog (5)	-0.657	-0.657	-17.084
frog (6)	0.953	0.953	-17.560
horse (7)	-0.394	-0.394	-7.833
ship (8)	8.019	8.019	**11.134**
truck (9)	3.787	3.787	4.604

Table 4.5: Experiment 1, ResNet20-Fixup (with bias): Verifying the effective hyperplanes' reconstruction for an adversarial example. Fig.4.8 illustrates the adversarial noise $\mathbf{Advr}$. Check Table 4.3 for details.

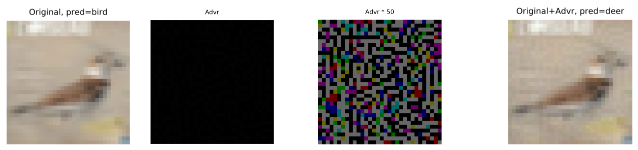


(a_1) $\mathbf{x}_0$ (a_2) $\mathbf{Advr}$ (a_3) Scaled a_2 (a_4) (a_1) + (a_2)

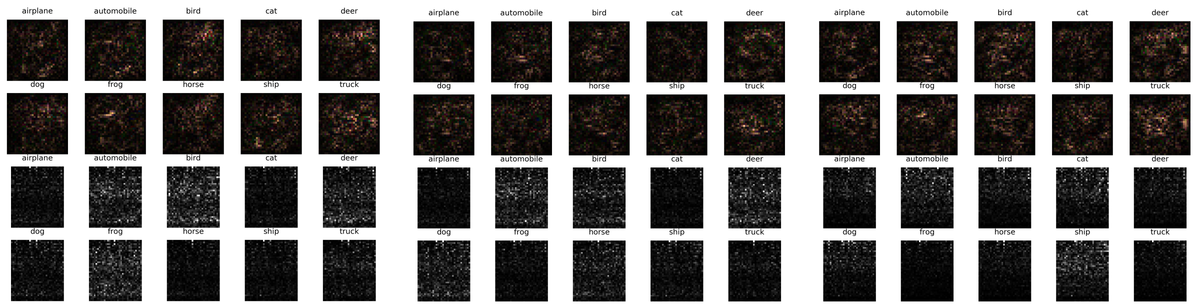


(b_1) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}$ (b_2) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0+\mathbf{Advr};\mathbf{x}_b]))\}$ (b_3) (b_2) - (b_1)

Figure 4.6: Experiment 1, VGG7 (with bias): Visualization of VGG7's effective hypersurfaces $\{\mathbf{H}_k^{Adj}\}_{k=0}^{9}$ with and without adversarial noise. (a_1): A "Bird" image. (a_2): An adversarial noise. (a_3): The scaled-up noise. (a_4): An adversarial example ($\mathbf{x}_0 + \mathbf{Advr}$) that fools VGG7 to predict "Cat". (b_1): The effective hypersurfaces for predicting 10 classes with the original "Bird" input. The first two rows illustrate the image parts of the hypersurfaces, $\{\mathbf{H}_k^{Adj,I}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}_{k=0}^{9}$ (A square represents $\mathbf{H}_k^{Adj,I}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))$ that has a shape of $32 \times 32 \times 3$); The last two rows present the bias parts, $\{\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}_{k=0}^{9}$ (A square visualizes a 384-d vector, $\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))$, being reshaped to 24×16). (b_2): The 10 effective hypersurfaces with (a_4) as the input. (b_3): The difference between with and without adversarial noise. The results are very interesting since the hypersurfaces themselves are very different, despite the appearance of the original (a_1) v.s. The adversarial image (a_4) is almost identical to the human eyes.

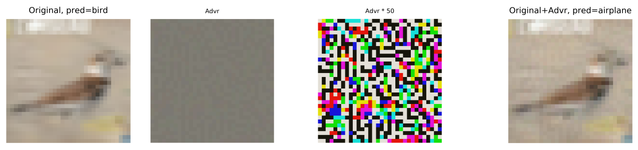


(a_1) $\mathbf{x}_0$ (a_2) $\mathbf{Advr}$ (a_3) Scaled a_2 (a_4) (a_1) + (a_2)

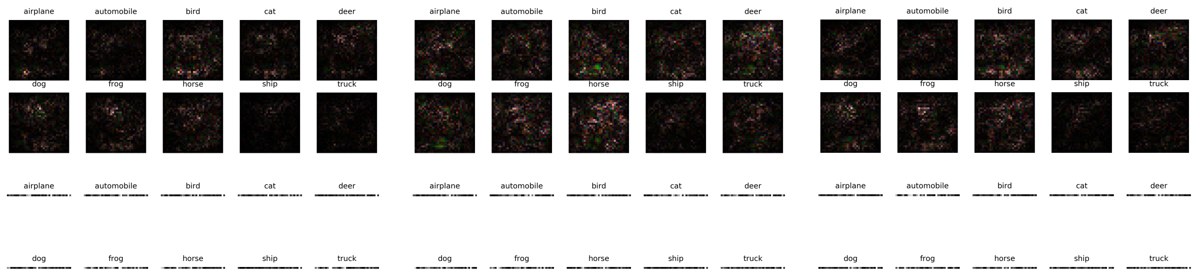


(b_1) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}$ (b_2) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0+\mathbf{Advr};\mathbf{x}_b]))\}$ (b_3) (b_2) - (b_1)

Figure 4.7: Experiment 1, ResNet20 (with bias): Visualization of ResNet20's effective hypersurfaces $\{\mathbf{H}_k^{Adj}\}_{k=0}^{9}$ with and without adversarial noise. The adversarial example (a_4) fools ResNet20 to predict "Deer". The last two rows of (b_1) present the bias parts of the effective hypersurfaces, $\{\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}_{k=0}^{9}$ (A square visualizes a 1152-d vector, $\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))$, being reshaped to 36×32). Check Fig.4.6 for other details.



(a_1) $\mathbf{x}_0$ (a_2) $\mathbf{Advr}$ (a_3) Scaled a_2 (a_4) (a_1) + (a_2)



(b_1) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}$ (b_2) $\{\mathbf{H}_k^{Adj}(\mathbf{z}([\mathbf{x}_0+\mathbf{Advr};\mathbf{x}_b]))\}$ (b_3) (b_2) - (b_1)

Figure 4.8: Experiment 1, ResNet20-Fixup (with bias): Visualization of ResNet20-Fixup's effective hypersurfaces $\{\mathbf{H}_k^{Adj}\}_{k=0}^{9}$ with and without adversarial noise. The adversarial example (a_4) fools ResNet20-Fixup to predict "Airplane'. The last two rows of (b_1) present the bias parts of the effective hypersurfaces, $\{\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))\}_{k=0}^{9}$ (A strip visualizes a 37-d vector, $\mathbf{H}_k^{Adj,b}(\mathbf{z}([\mathbf{x}_0;\mathbf{x}_b]))$). Check Fig.4.6 for other details.

[3] (A targeted attack method). We name these samples as $\{\mathbf{Advr}_m \mid m \in \{0, ..., 9\}, \mathbf{Advr}_2 = \theta\}$ where adding $\mathbf{Advr}_m$ to the "Bird" can fool the model from the label to the class index m, and $\mathbf{Advr}_2 = \theta$ indicates no adversarial noise (zero noise). We illustrate the effective hypersurface's response to these samples in Fig.4.9, 4.10, 4.11. Also, we project the image parts and the bias parts of generated hyperplanes to a 3-d space, respectively, using tSNE to see their dispersion. For comparison, 10 more images ($x_1, ..., x_{10}$) of different classes (not "Bird") are selected from the test set.

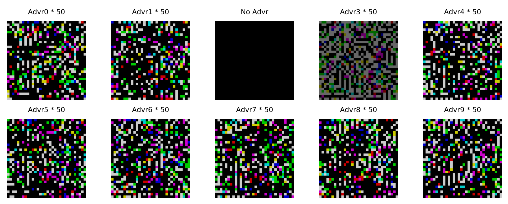
4.3.3 Experiment 3: Comparing the spread of hypersurfaces in response to adversarial vs. Gaussian noise

Given a CNN model (VGG7/ResNet20/ResNet20-Fixup), similar to Section 4.2.3, we loop a β from 0.0 to 1.0 and scale down the 9 adversarial samples from Experiment 2 (Section 4.3.2), i.e., $\beta \times \mathbf{Advr}_m$, to prepare 50 adversarial patterns that can deceive the CNN model and result in misclassification of the "Bird" image. At the same time, we generate 50 Gaussian noise patterns ($32 \times 32 \times 3$) using the same pixel mean and variance as those adversarial ones. These Guassian noise patterns, when added to the "Bird", will not affect the CNN model on predicting the "Bird" image.

We use Factor Analysis to project image parts and bias parts of hyperplanes, generated from the label-index ($k = 2$) hypersurface by taking the 100 noise patterns as inputs, to low-dimensional spaces. We visualize the results in Fig.4.12, 4.13, 4.14.

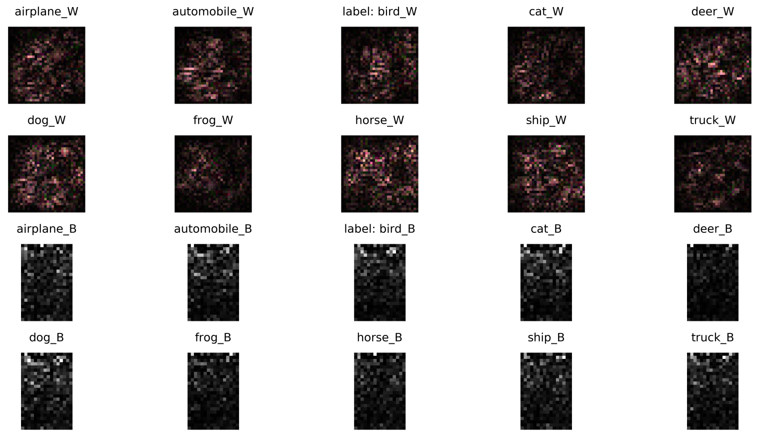
4.3.4 Analysis

These three experiments reveal that CNN suffers from its brittle decision hypersurfaces. Experiment 1 shows that the effective hyperplanes under an adversarial attack ((b_2) of Fig.4.6, 4.7, 4.8) significantly differ from the ones of the original image ((b_1) of Fig.4.6, 4.7, 4.8). Even for a single hypersurface, different adversarial noise samples can easily result in visible fluctuations in Experiment 2 ((b_1) of Fig.4.9, Fig.4.10, Fig.4.11); These differences are geometrically measured through large distances among points of their (c_1) and (c_2) figures. In addition, Experiment 3 reveals that

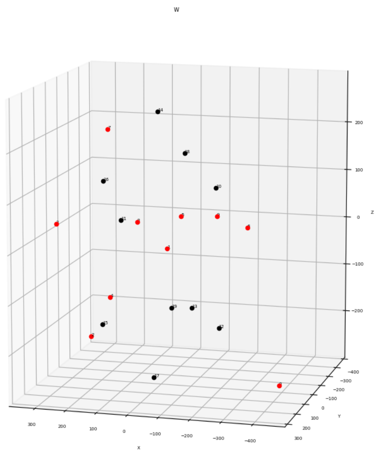


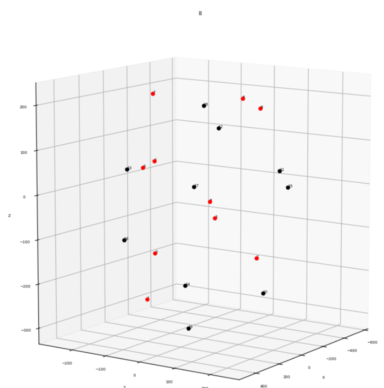
(a_1) Scaled $\{\mathbf{Advr}_m\}_{m=0}^{9}$

(a_2) $\{\mathbf{x}_0 + \mathbf{Advr}_m\}_{m=0}^{9}$



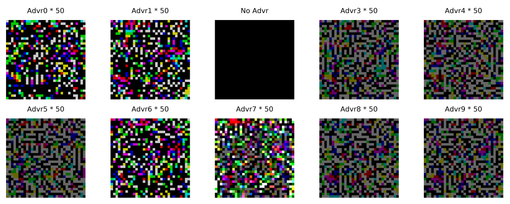
(b_1) $\{\mathbf{H}_{k=2}^{Adj}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}_m; \mathbf{x}_b]))\}_{m=0}^{9}$

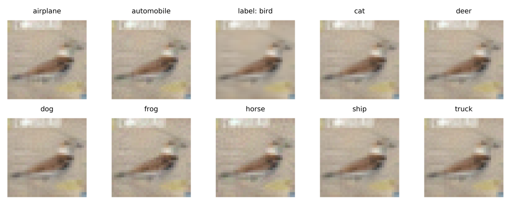




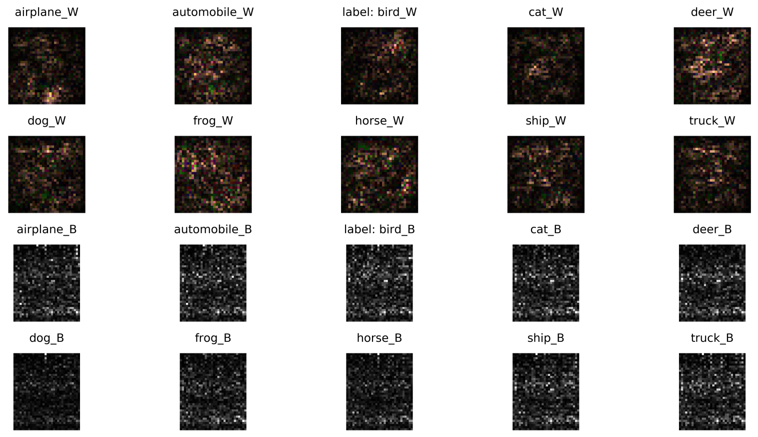
(c_1) tSNE of the first two rows of $(b_1)$$(c_2)$ tSNE of the last two rows of (b_1)

Figure 4.9: Experiment 2, VGG7 (with bias): Visualize the effective hypersurface for the "Bird" unit in response to different adversarial noise. $k = 2$ denotes the "Bird" index. (a_1): The 9 scaled-up adversarial noise samples (for visualization purposes) $\{\mathbf{Advr}_m \mid m \in \{0, ..., 9\}, \mathbf{Advr}_2 = \theta\}$ generated from targeted attacks. Each $\mathbf{Advr}_m$, when added to the "Bird", can fool VGG7 to the specific class index m; $\mathbf{Advr}_2$ does not have noise. (a_2): The sums of (a_1) and the original "Bird" image, the adversarial inputs. The subtitle "label:bird" depicts the original $\mathbf{x}_0$. (b_1): The effective hypersurface for the "Bird" unit in response to the inputs from (a_2). The first two rows illustrate the image part $\mathbf{H}_{k=2}^{Adj,I}$; The last two rows illustrate the bias part $\mathbf{H}_{k=2}^{Adj,b}$. The "label:bird" illustrates the hypersurface's response without adversarial noise. (c_1): tSNE projection of the image parts (Red), $\{\mathbf{H}_{k=2}^{Adj,I}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}_m; \mathbf{x}_b]))\}_{m=0}^{9}$, to a 3-d space. (c_2): tSNE projection of the bias parts (Red), $\{\mathbf{H}_{k=2}^{Adj,b}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}_m; \mathbf{x}_b]))\}_{m=0}^{9}$, to a 3-d space. Black dots are corresponding parts of effective hyperplanes generated from 10 random images from different classes (not "Bird"). Note that the spread due to adversarial noise (Red) is almost as broad as that due to inputs from different classes. Check Section 4.3 for details.

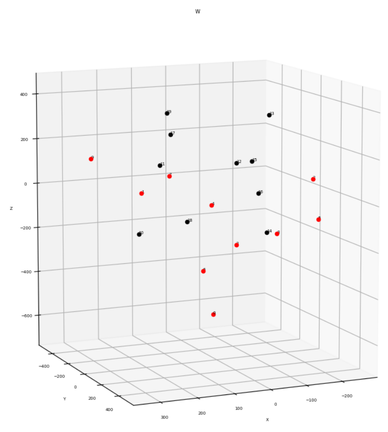


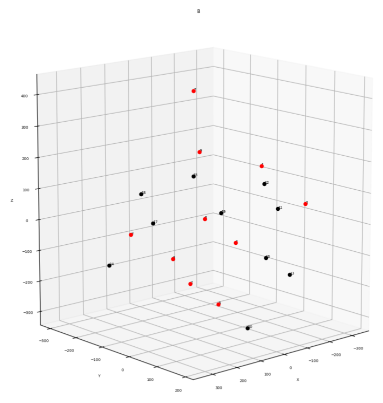


(a_1) Scaled $\{\mathbf{Advr}_m\}_{m=0}^{9}$ (a_2) $\{\mathbf{x}_0 + \mathbf{Advr}_m\}_{m=0}^{9}$



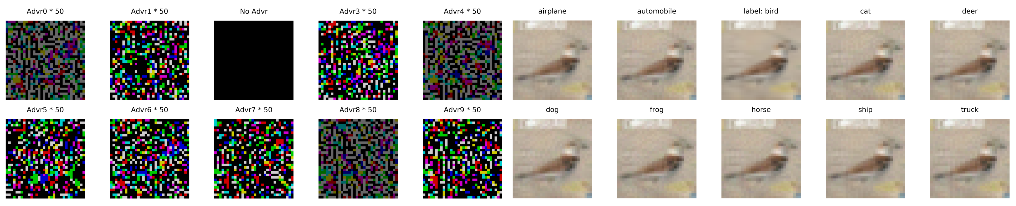
(b_1) $\{\mathbf{H}_{k=2}^{Adj}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}_m; \mathbf{x}_b]))\}_{m=0}^{9}$



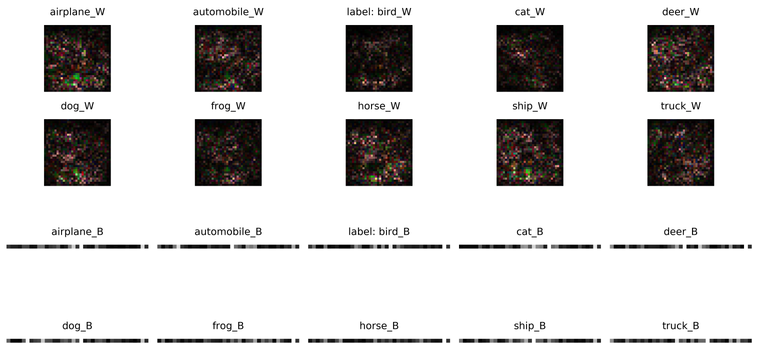


(c_1) tSNE of the first two rows of $(b_1)$$(c_2)$ tSNE of the last two rows of (b_1)

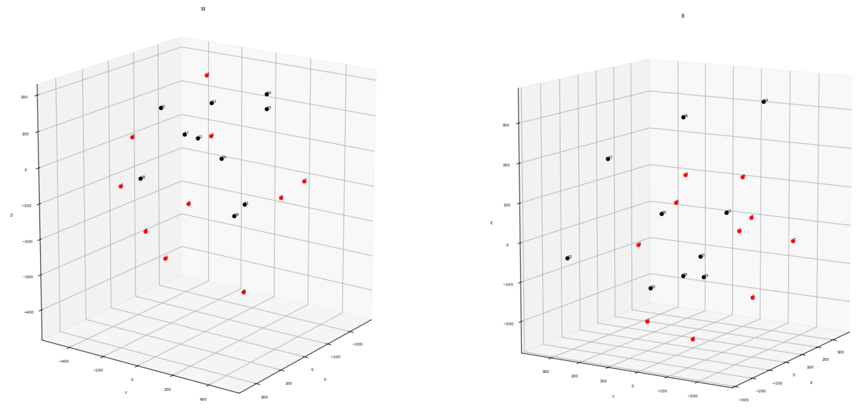
Figure 4.10: Experiment 2, ResNet20 (with bias): Visualize the effective hypersurface for the "Bird" unit in response to different adversarial noise. Each $\mathbf{Advr}_m$ in (a_1), when added to the "Bird", can fool ResNet20 to another class. Check Fig.4.9 for details.



(a_1) Scaled $\{\mathbf{Advr}_m\}_{m=0}^{9}$ (a_2) $\{\mathbf{x}_0 + \mathbf{Advr}_m\}_{m=0}^{9}$

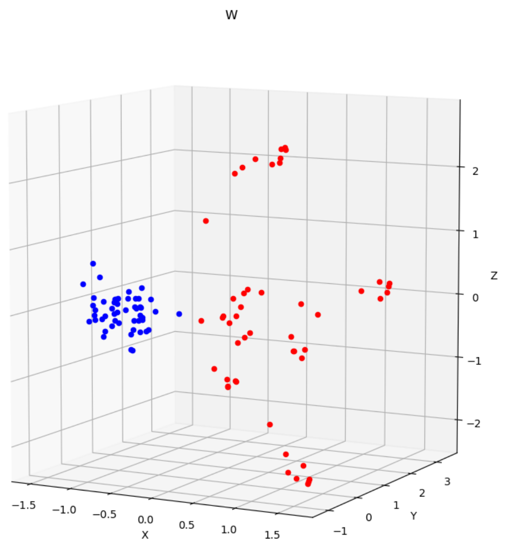


(b_1) $\{\mathbf{H}_{k=2}^{Adj}(\mathbf{z}([\mathbf{x}_0 + \mathbf{Advr}_m; \mathbf{x}_b]))\}_{m=0}^{9}$

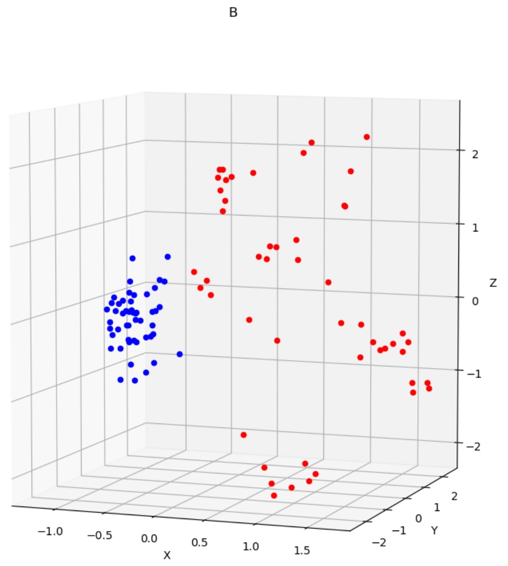


(c_1) tSNE of the first two rows of $(b_1)$$(c_2)$ tSNE of the last two rows of (b_1)

Figure 4.11: Experiment 2, ResNet20-Fixup (with bias): Visualize the effective hypersurface for the "Bird" unit in response to different adversarial noise. Each $\mathbf{Advr}_m$ in (a_1), when added to the "Bird", can fool ResNet20-Fixup to another class. Check Fig.4.9 for details.

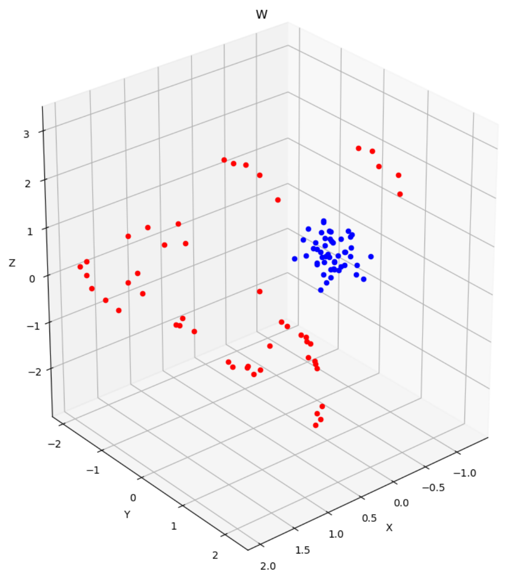


(a_1) Image parts of hyperplanes

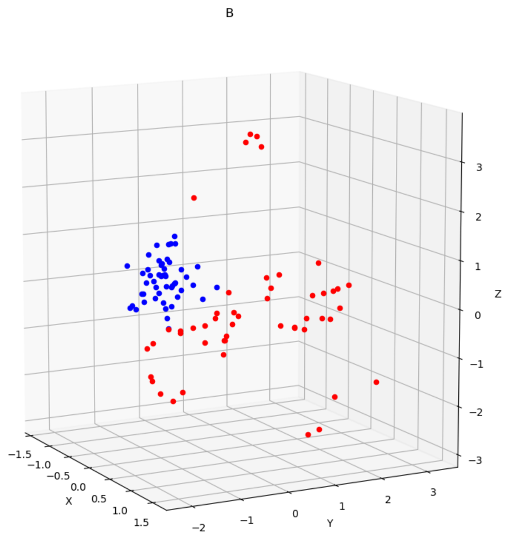


(a_2) Bias parts of hyperplanes

Figure 4.12: Experiment 3, VGG7 (with bias): Comparing the spread of hypersurfaces in response to adversarial vs. Gaussian noise using Factor Analysis. Similar to Fig.4.5: (**Red**) 50 effective hyperplanes computed from 50 adversarial examples (original image plus adversarial noise patterns); (**Blue**) 50 effective hyperplanes computed from 50 non-adversarial examples (original image plus Gaussian noise patterns). Check Section 4.3.3 for details.

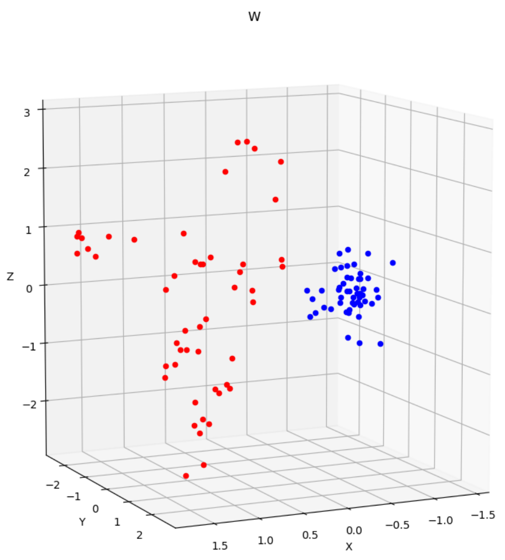


(a_1) Image parts of hyperplanes

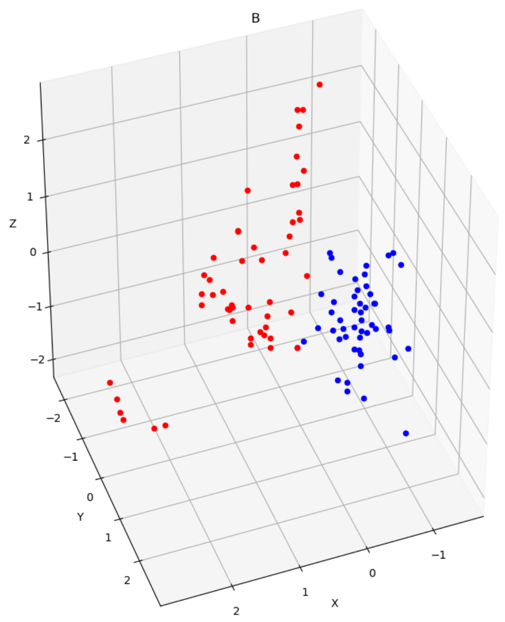


(a_2) Bias parts of hyperplanes

Figure 4.13: Experiment 3, ResNet20 (with bias): Comparing the spread of hypersurfaces in response to adversarial vs. Gaussian noise using Factor Analysis. Color notations are the same as Fig.4.12.



(a_1) Image parts of hyperplanes



(a_2) Bias parts of hyperplanes

Figure 4.14: Experiment 3, ResNet20-Fixup (with bias): Comparing the spread of hypersurfaces in response to adversarial vs. Gaussian noise using Factor Analysis. Color notations are the same as Fig.4.12.

despite adversarial noise patterns being imperceptible to human eyes, they can statistically result in more considerable variations than the Gaussian patterns, although they share the same means and variances. Therefore, we conclude that adversarial examples can easily deceive CNN due to the brittleness of CNN's decision.

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The main goal of this dissertation is to investigate how a CNN works on input images by using a mathematical analysis framework we introduced: AdjointBackMap. This framework circumvents the difficulty of finding the inverse by mapping every convolution kernel (in higher layers) or weight vector (in an FC layer) back to the input image space with the help of adjoint operators. Given an arbitrary unit inside a CNN, our algorithm can reconstruct an effective hyperplane that, when multiplied by the original input (through an inner product), will precisely replicate the output value of the unit. Such an effective hyperplane accurately summarizes the CNN's decision process from the input end all the way up to the unit being considered. We divided our study into two parts: CNN without bias and CNN with bias. For each part, we also applied our framework to analyze adversarial attacks.

5.1.1 Model a CNN without bias

To simplify the problem, we first consider the normed space only consisting of all input images to model CNN without bias. Our experiments showed that,

1. All effective hyperplanes reconstructed from high-level kernels are not human-recognizable patterns. Since these patterns determine a high-level feature map or FC layer output, it suggests that CNN's decision process is not like that of human vision, unlike what other interpretation approaches [15, 16] might suggest.

2. CNN's decision process is largely conditioned on the current input image and is extremely sensitive to adversarial noise. These observations explain why adversarial examples can deceive CNNs because two effective hyperplanes of a CNN corresponding to two human-indistinguishable images can be very different from each other.

5.1.2 Model a CNN with bias

Next, we extend the normed space to include the bias from all trained CNN layers as another input component. This way, we can also support CNN models that use batch normalization. Based on this extended normed space, we upgrade our adjoint-operator-based algorithm to AdjointBackMapV2, which maps high-level weights back to the extended space to reconstruct an effective hypersurface. Unlike the previous reconstructions in AdjointBackMap, any effective hypersurface in this new version is composed of two parts: the part for the input image and the part for the bias. Our experiments showed the following,

1. AdjointBackMapV2 achieves near 0 reconstruction error with three prevalent CNN models (VGG7/ResNet20/ResNet20-Fixup, using either batch normalization or conventional bias) on the CIFAR-10 and CIFAR-100 datasets.

2. Both image and bias parts of effective hypersurfaces reconstructed for units in the FC layer are still very sensitive to adversarial noise. Since these effective hypersurfaces summarize CNN's decision process, the results strengthen our previous point that adversarial examples can easily deceive CNN due to the brittleness of CNN's decision.

5.2 Discussion and future work

Studying CNN's inner workings is critical research for understanding such ”Black-box” models. Despite many efforts in making CNNs interpretable, as we discussed in Section 1.1, adversarial examples still hinder model explanation since two images that share visually similar features might be very difficult for a CNN model to distinguish, and even deceive the interpretable explanations. Our results conclude that this is because CNN's robustness issues may result from the brittleness of their effective hypersurfaces. It suggests that adjusting CNN's architecture to reduce the sensitivity of effective hypersurfaces might be a promising direction to overcome this weakness. In addition, it would be worth extending our theory to other domains, like NLP (Natural Language Processing), to see how a model there makes its decisions. In general, we expect our method will help gain deeper insights into the hidden mechanisms of deep neural networks.

REFERENCES

[1] H. Zhang, Y. N. Dauphin, and T. Ma, “Fixup initialization: Residual learning without normalization,” *arXiv preprint arXiv:1901.09321*, 2019.

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” *arXiv preprint arXiv:1412.6572*, 2014.

[3] A. Kurakin, I. Goodfellow, and S. Bengio, “Adversarial examples in the physical world,” *arXiv preprint arXiv:1607.02533*, 2016.

[4] M. Lin, Q. Chen, and S. Yan, “Network in network,” *arXiv preprint arXiv:1312.4400*, 2013.

[5] Q. Wan and Y. Choe, “Adjointbackmap: Reconstructing effective decision hypersurfaces from cnn layers using adjoint operators,” *arXiv preprint arXiv:2012.09020*, 2020.

[6] D. H. Hubel and T. N. Wiesel, “Receptive fields and functional architecture of monkey striate cortex,” *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[7] K. Fukushima, “Neocognitron: A hierarchical neural network capable of visual pattern recognition,” *Neural networks*, vol. 1, no. 2, pp. 119–130, 1988.

[8] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[9] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, “Deep, big, simple neural nets for handwritten digit recognition,” *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[11] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[13] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[14] D. Matthew and R. Fergus, “Visualizing and understanding convolutional neural networks,” in *Proceedings of the 13th European Conference Computer Vision and Pattern Recognition, Zurich, Switzerland*, pp. 6–12, 2014.

[15] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, “Learning deep features for discriminative localization,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2921–2929, 2016.

[16] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-cam: Visual explanations from deep networks via gradient-based localization,” in *Proceedings of the IEEE international conference on computer vision*, pp. 618–626, 2017.

[17] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for simplicity: The all convolutional net,” *arXiv preprint arXiv:1412.6806*, 2014.

[18] K. Xu, G. Zhang, S. Liu, Q. Fan, M. Sun, H. Chen, P.-Y. Chen, Y. Wang, and X. Lin, “Adversarial t-shirt! evading person detectors in a physical world,” *arXiv*, pp. arXiv–1910, 2019.

[19] J. Heo, S. Joo, and T. Moon, “Fooling neural network interpretations via adversarial model manipulation,” *Advances in Neural Information Processing Systems*, vol. 32, pp. 2925–2936, 2019.

[20] A.-K. Dombrowski, M. Alber, C. J. Anders, M. Ackermann, K.-R. Müller, and P. Kessel, "Explanations can be manipulated and geometry is to blame," *arXiv preprint arXiv:1906.07983*, 2019.

[21] A. Subramanya, V. Pillai, and H. Pirsiavash, "Fooling network interpretation in image classification," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2020–2029, 2019.

[22] A. Dosovitskiy and T. Brox, "Inverting visual representations with convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4829–4837, 2016.

[23] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[24] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144, 2016.

[25] P. Dabkowski and Y. Gal, "Real time image saliency for black box classifiers," in *Advances in Neural Information Processing Systems*, pp. 6967–6976, 2017.

[26] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," *arXiv preprint arXiv:1704.02685*, 2017.

[27] L. M. Zintgraf, T. S. Cohen, T. Adel, and M. Welling, "Visualizing deep neural network decisions: Prediction difference analysis," *arXiv preprint arXiv:1702.04595*, 2017.

[28] B. Kim, M. Wattenberg, J. Gilmer, C. Cai, J. Wexler, F. Viegas, *et al.*, "Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav)," in *International conference on machine learning*, pp. 2668–2677, 2018.

[29] V. Petsiuk, A. Das, and K. Saenko, "Rise: Randomized input sampling for explanation of black-box models," *arXiv preprint arXiv:1806.07421*, 2018.

[30] M. Ibrahim, M. Louie, C. Modarres, and J. Paisley, "Global explanations of neural networks: Mapping the landscape of predictions," in *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 279–287, 2019.

[31] M. Ancona, C. Öztireli, and M. Gross, "Explaining deep neural networks with a polynomial time algorithm for shapley values approximation," *arXiv preprint arXiv:1903.10992*, 2019.

[32] C. Olah, A. Mordvintsev, and L. Schubert, "Feature visualization," *Distill*, vol. 2, no. 11, p. e7, 2017.

[33] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 4905–4913, 2016.

[34] A. Ghorbani, A. Abid, and J. Zou, "Interpretation of neural networks is fragile," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3681–3688, 2019.

[35] T. Wiatowski and H. Bölcskei, "A mathematical theory of deep convolutional neural networks for feature extraction," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1845–1866, 2017.

[36] Q. Qiu, X. Cheng, G. Sapiro, *et al.*, "Dcfnet: Deep neural network with decomposed convolutional filters," in *International Conference on Machine Learning*, pp. 4198–4207, PMLR, 2018.

[37] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: Convergence and generalization in neural networks," *arXiv preprint arXiv:1806.07572*, 2018.

[38] S. Hayou, A. Doucet, and J. Rousseau, "Mean-field behaviour of neural tangent kernel for deep neural networks," *arXiv preprint arXiv:1905.13654*, 2019.

[39] W. Rudin, "Functional analysis," 1973.

[40] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[41] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. icml*, vol. 30, p. 3, Citeseer, 2013.

[42] S. Banach, *Theory of linear operations*. Elsevier, 1987.

[43] D. G. Luenberger, *Optimization by vector space methods*. John Wiley & Sons, 1997.

[44] G. B. Folland, *Real analysis: modern techniques and their applications*, vol. 40. John Wiley & Sons, 1999.

[45] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[46] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[47] H. Zhang, “Fixup initialization implementation in pytorch.” `https://github.com/hongyi-zhang/Fixup`, Nov, 2020 (accessed).

[48] H. Wang, X. Wu, Z. Huang, and E. P. Xing, “High-frequency component helps explain the generalization of convolutional neural networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8684–8694, 2020.

[49] Q. Wan and Y. Choe, “Adjointbackmapv2: Precise reconstruction of arbitrary cnn unit’s activation via adjoint operators,” *arXiv preprint arXiv:2110.01736*, 2021.

[50] S. Ioffe, “Batch renormalization: Towards reducing minibatch dependence in batch-normalized models,” *arXiv preprint arXiv:1702.03275*, 2017.

[51] G. O. GitHub, “Tensorflow official models.” `https://github.com/tensorflow/models`, Nov, 2018 (accessed).

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

APPENDIX A

APPENDIX FOR MODELING CNN WITHOUT BIAS

A.1 Hardware and software for verification experiments

We built TensorFlow 1.15.4 from the source code and enabled its functionality on AVX-2, AVX-512, FMA3 instruction sets to speed up all experiments. The experiments were done on an Intel 9940X CPU (with 128GB DRAM) or an Intel 10920X CPU (with 256GB DRAM).

A.2 Tables

Layer	Parameters	Out-channel Feature Maps
Conv0	$3 \times 3 \times 3 \times 32$	N/S
ReLU0	N/A	N/S
Conv1	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
ReLU1	N/A	N/S
Avg-pool-by-2	N/A	N/S
Conv2	$3 \times 3 \times 32 \times 64$	$16 \times 16 \times 64 (= 16,384)$
ReLU2	N/A	N/S
Conv3	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
ReLU3	N/A	N/S
Avg-pool-by-2	N/A	N/S
Conv4	$3 \times 3 \times 64 \times 96$	$8 \times 8 \times 96 (= 6,144)$
ReLU4	N/A	N/S
Conv5	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
ReLU5	N/A	N/S
Global-pool (g_p)	N/A	N/S
FC	[96, 10]	10
ReLU6	N/A	N/S

Avg-pool-by-2 denotes average pooling with a window size of 3 and stride size of 2;
N/A denotes no learnable parameters;
N/S denotes it is not necessary for our method.

Table A.1: Parameters in VGG7.

Block (shortcut)	Layer	Parameters	Out-channel Feature Maps
	Conv0	$3 \times 3 \times 3 \times 32$	N/S
	ReLU0	N/A	N/S
Residual 0 (identity)	Conv1	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU1	N/A	N/S
	Conv2	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU2	N/A	N/S
Residual 1 (identity)	Conv3	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU3	N/A	N/S
	Conv4	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU4	N/A	N/S
Residual 2 (identity)	Conv5	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU5	N/A	N/S
	Conv6	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
	ReLU6	N/A	N/S
Residual 3 (avg-pool+pad)	Conv7 (stride=2)	$3 \times 3 \times 32 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU7	N/A	N/S
	Conv8	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU8	N/A	N/S
Residual 4 (identity)	Conv9	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU9	N/A	N/S
	Conv10	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU10	N/A	N/S
Residual 5 (identity)	Conv11	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU11	N/A	N/S
	Conv12	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
	ReLU12	N/A	N/S
Residual 6 (avg-pool+pad)	Conv13 (stride=2)	$3 \times 3 \times 64 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU13	N/A	N/S
	Conv14	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU14	N/A	N/S
Residual 7 (identity)	Conv15	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU15	N/A	N/S
	Conv16	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU16	N/A	N/S
Residual 8 (identity)	Conv17	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU17	N/A	N/S
	Conv18	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
	ReLU18	N/A	N/S
	Global-pool (g_p)	N/A	N/S
	FC	96×10	10
	ReLU19	N/A	N/S

avg-pool+pad denotes average pooling with a window size of 1 and stride size of 2, and padding zero channels to match the quantity of out channels for summation; weights rescaling is used after Conv2, 4, 6, 8, 10, 12, 14, 16, 18 but not listed here.

Table A.2: Parameters in Fixup-ResNet20 (refer to table A.1).

Layer	$RM4$	$RM3$	$RM2$	$RM1$	$RM0$
Conv0	N/A	N/A	N/A	N/A	N/A
Conv1	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv2	$d_{in} \times 16 \times 16 \times 32 \times 64$	$d_{in} \times 32 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv3	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv4	$d_{in} \times 8 \times 8 \times 64 \times 96$	$d_{in} \times 64 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv5	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
FC	N/A	N/A	N/A	N/A	$d_{in} \times 10$

d_{in} denotes $32 \times 32 \times 3$;
N/A denotes a layer where our AdjointBackMap is not applicable.

Table A.3: Dimensions of $\mathbf{H}^{Adj}(\mathbf{z}(\mathbf{x}))$ with different RMs On VGG7.

Layer	$RM4$	$RM3$	$RM2$	$RM1$	$RM0$
Conv0	N/A	N/A	N/A	N/A	N/A
Conv1	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv2	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv3	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv4	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv5	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv6	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv7	$d_{in} \times 16 \times 16 \times 32 \times 64$	$d_{in} \times 32 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv8	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv9	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv10	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv11	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv12	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv13	$d_{in} \times 8 \times 8 \times 64 \times 96$	$d_{in} \times 64 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv14	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv15	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv16	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv17	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv18	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
FC	N/A	N/A	N/A	N/A	$d_{in} \times 10$

d_{in} denotes $32 \times 32 \times 3$;
N/A denotes a layer where our AdjointBackMap is not applicable.

Table A.4: Dimensions of $\mathbf{H}^{Adj}(\mathbf{z}(\mathbf{x}))$ with different RMs on Fixup-ResNet20.

APPENDIX B

APPENDIX FOR MODELING CNN WITH BIAS

B.1 Hardware and software for verification experiments

Verification experiments (figures 3.6, 3.7, 3.8, 3.9, 3.10, 3.11) were conducted on Intel 10920X (VGG7/ResNet20) and 9940X (ResNet20-Fixup) CPUs. Both set up ran TensorFlow 1.15.4 with AVX-2, AVX-512, and FMA3 instruction sets enabled (built from source, the same as Section A.1).

B.2 Tables

Original Layer	Equivalent Layer	Parameters	Out-ch Feature Maps
Input ($\mathbf{x}_n$)	Input ($[\mathbf{x}_n; \mathbf{x}_b]$)	N/A	N/A
Conv0 ($\mathbf{w}_0$)	Conv0 ($\mathbf{w}_0^{'}$)	N/A	N/A
BN0	B0 ($\mathbf{x}_b[\mathbf{b}_0^{'}]$)	N/A	N/A
ReLU0		N/A	N/A
Conv1 ($\mathbf{w}_1$)	Conv1 ($\mathbf{w}_1^{'}$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (= 32,768)$
BN1	B1 ($\mathbf{x}_b[\mathbf{b}_1^{'}]$)	N/A	N/A
ReLU1		N/A	N/A
Avg-pool-2		N/A	N/A
Conv2 ($\mathbf{w}_2$)	Conv2 ($\mathbf{w}_2^{'}$)	$3 \times 3 \times 32 \times 64$	$16 \times 16 \times 64 (= 16,384)$
BN2	B2 ($\mathbf{x}_b[\mathbf{b}_2^{'}]$)	N/A	N/A
ReLU2		N/A	N/A
Conv3 ($\mathbf{w}_3$)	Conv3 ($\mathbf{w}_3^{'}$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (= 16,384)$
BN3	B3 ($\mathbf{x}_b[\mathbf{b}_3^{'}]$)	N/A	N/A
ReLU3		N/A	N/A
Avg-pool-2		N/A	N/A
Conv4 ($\mathbf{w}_4$)	Conv4 ($\mathbf{w}_4^{'}$)	$3 \times 3 \times 64 \times 96$	$8 \times 8 \times 96 (= 6,144)$
BN4	B4 ($\mathbf{x}_b[\mathbf{b}_4^{'}]$)	N/A	N/A
ReLU4		N/A	N/A
Conv5 ($\mathbf{w}_5$)	Conv5 ($\mathbf{w}_5^{'}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (= 6,144)$
BN5	B5 ($\mathbf{x}_b[\mathbf{b}_5^{'}]$)	N/A	N/A
ReLU5		N/A	N/A
global-pool (g_p)		N/A	N/A
FC ($\mathbf{w}_{fc}$)		$96 \times 10/100$	$10/100$
ReLU6		N/A	N/A

$\mathbf{w}_i^{'}$ and $\mathbf{b}_i^{'}$ are computed using Eq.(3.6);
$\mathbf{x}_b$ is a sequentical concatenation of $\mathbf{b}_0^{'}$~$\mathbf{b}_5^{'}$;
Avg-pool-2: average pooling with window size of 3 and stride of 2;
N/A: it is not necessary for our method.

Table B.1: Parameters for VGG7.

Equivalent Layer	RM_4	RM_3	RM_2	RM_1	RM_0
Conv0	N/A	N/A	N/A	N/A	N/A
Conv1	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv2	$d_{in} \times 16 \times 16 \times 32 \times 64$	$d_{in} \times 32 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv3	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv4	$d_{in} \times 8 \times 8 \times 64 \times 96$	$d_{in} \times 64 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv5	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
FC	N/A	N/A	N/A	N/A	$d_{in} \times 10/100$

$d_{in} = (32 \times 32 \times 3 + 384)$ where $384 = (32 + 64 + 96) \times 2$;
N/A: it is not necessary for our method.

Table B.2: Dimensions of $\mathbf{H}^{Adj}(\frac{[\mathbf{x}_n;\mathbf{x}_b]}{8})$ with different RMs on VGG7.

Equivalent layer	RM_4	RM_3	RM_2	RM_1	RM_0
Conv0	N/A	N/A	N/A	N/A	N/A
Conv1	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv2	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv3	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv4	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv5	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv6	$d_{in} \times 32 \times 32 \times 32 \times 32$	$d_{in} \times 32 \times 32$	$d_{in} \times 32 \times 32 \times 32$	$d_{in} \times 32$	N/A
Conv7	$d_{in} \times 16 \times 16 \times 32 \times 64$	$d_{in} \times 32 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv8	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv9	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv10	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv11	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv12	$d_{in} \times 16 \times 16 \times 64 \times 64$	$d_{in} \times 64 \times 64$	$d_{in} \times 16 \times 16 \times 64$	$d_{in} \times 64$	N/A
Conv13	$d_{in} \times 8 \times 8 \times 64 \times 96$	$d_{in} \times 64 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv14	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv15	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv16	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv17	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
Conv18	$d_{in} \times 8 \times 8 \times 96 \times 96$	$d_{in} \times 96 \times 96$	$d_{in} \times 8 \times 8 \times 96$	$d_{in} \times 96$	N/A
FC	N/A	N/A	N/A	N/A	$d_{in} \times 10/100$

For ResNet20: $d_{in} = (32 \times 32 \times 3 + 1,152)$ where $1,152 = (32 + 64 + 96) \times 6$;
For ResNet20-Fixup: $d_{in} = (32 \times 32 \times 3 + 37)$.

Table B.3: Dimensions of $\mathbf{H}^{Adj}(\frac{[\mathbf{x}_n;\mathbf{x}_b]}{8})$ with different RMs on ResNet20/ResNet20-Fixup.

Block (shortcut)	Original layer	Equivalent layer	Parameters	Out-ch feature maps
	Input ($\mathbf{x}_n$)	Input ($[\mathbf{x}_n; \mathbf{x}_b]$)	N/A	N/A
	Conv0 ($\mathbf{w}_0$)	Conv0 ($\mathbf{w}_0'$)	N/A	N/A
	BN0	B0 ($\mathbf{x}_b[\mathbf{b}_0']$)	N/A	N/A
	Leaky_ReLU0		N/A	N/A
Residual 0 (identity)	Conv1 ($\mathbf{w}_1$)	Conv1 ($\mathbf{w}_1'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN1	B1 ($\mathbf{x}_b[\mathbf{b}_1']$)	N/A	N/A
	Leaky_ReLU1		N/A	N/A
	Conv2 ($\mathbf{w}_2$)	Conv2 ($\mathbf{w}_2'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN2	B2 ($\mathbf{x}_b[\mathbf{b}_2']$)	N/A	N/A
	Leaky_ReLU2		N/A	N/A
Residual 1 (identity)	Conv3 ($\mathbf{w}_3$)	Conv3 ($\mathbf{w}_3'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN3	B3 ($\mathbf{x}_b[\mathbf{b}_3']$)	N/A	N/A
	Leaky_ReLU3		N/A	N/A
	Conv4 ($\mathbf{w}_4$)	Conv4 ($\mathbf{w}_4'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN4	B4 ($\mathbf{x}_b[\mathbf{b}_4']$)	N/A	N/A
	Leaky_ReLU4		N/A	N/A
Residual 2 (identity)	Conv5 ($\mathbf{w}_5$)	Conv5 ($\mathbf{w}_5'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN5	B5 ($\mathbf{x}_b[\mathbf{b}_5']$)	N/A	N/A
	Leaky_ReLU5		N/A	N/A
	Conv6 ($\mathbf{w}_6$)	Conv6 ($\mathbf{w}_6'$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32 (32,768)$
	BN6	B6 ($\mathbf{x}_b[\mathbf{b}_6']$)	N/A	N/A
	Leaky_ReLU6		N/A	N/A
Residual 3 (avg-pool+pad)	Conv7 ($\mathbf{w}_7$, $s = 2$)	Conv7 ($\mathbf{w}_7'$, $s = 2$)	$3 \times 3 \times 32 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN7	B7 ($\mathbf{x}_b[\mathbf{b}_7']$)	N/A	N/A
	Leaky_ReLU7		N/A	N/A
	Conv8 ($\mathbf{w}_8$)	Conv8 ($\mathbf{w}_8'$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN8	B8 ($\mathbf{x}_b[\mathbf{b}_8']$)	N/A	N/A
	Leaky_ReLU8		N/A	N/A
Residual 4 (identity)	Conv9 ($\mathbf{w}_9$)	Conv9 ($\mathbf{w}_9'$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN9	B9 ($\mathbf{x}_b[\mathbf{b}_9']$)	N/A	N/A
	Leaky_ReLU9		N/A	N/A
	Conv10 ($\mathbf{w}_{10}$)	Conv10 ($\mathbf{w}_{10}'$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN10	B10 ($\mathbf{x}_b[\mathbf{b}_{10}']$)	N/A	N/A
	Leaky_ReLU10		N/A	N/A
Residual 5 (identity)	Conv11 ($\mathbf{w}_{11}$)	Conv11 ($\mathbf{w}_{11}'$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN11	B11 ($\mathbf{x}_b[\mathbf{b}_{11}']$)	N/A	N/A
	Leaky_ReLU11		N/A	N/A
	Conv12 ($\mathbf{w}_{12}$)	Conv12 ($\mathbf{w}_{12}'$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64 (16,384)$
	BN12	B12 ($\mathbf{x}_b[\mathbf{b}_{12}']$)	N/A	N/A
	Leaky_ReLU12		N/A	N/A
Residual 6 (avg-pool+pad)	Conv13 ($\mathbf{w}_{13}$, $s = 2$)	Conv13 ($\mathbf{w}_{13}'$, $s = 2$)	$3 \times 3 \times 64 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN13	B13 ($\mathbf{x}_b[\mathbf{b}_{13}']$)	N/A	N/A
	Leaky_ReLU13		N/A	N/A
	Conv14 ($\mathbf{w}_{14}$)	Conv14 ($\mathbf{w}_{14}'$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN14	B14 ($\mathbf{x}_b[\mathbf{b}_{14}']$)	N/A	N/A
	Leaky_ReLU14		N/A	N/A
Residual 7 (identity)	Conv15 ($\mathbf{w}_{15}$)	Conv15 ($\mathbf{w}_{15}'$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN15	B15 ($\mathbf{x}_b[\mathbf{b}_{15}']$)	N/A	N/A
	Leaky_ReLU15		N/A	N/A
	Conv16 ($\mathbf{w}_{16}$)	Conv16 ($\mathbf{w}_{16}'$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN16	B16 ($\mathbf{x}_b[\mathbf{b}_{16}']$)	N/A	N/A
	Leaky_ReLU16		N/A	N/A
Residual 8 (identity)	Conv17 ($\mathbf{w}_{17}$)	Conv17 ($\mathbf{w}_{17}'$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN17	B17 ($\mathbf{x}_b[\mathbf{b}_{17}']$)	N/A	N/A
	Leaky_ReLU17		N/A	N/A
	Conv18 ($\mathbf{w}_{18}$)	Conv18 ($\mathbf{w}_{18}'$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96 (6,144)$
	BN18	B18 ($\mathbf{x}_b[\mathbf{b}_{18}']$)	N/A	N/A
	Leaky_ReLU18		N/A	N/A
	g_p		N/A	N/A
	FC ($\mathbf{w}_{fc}$)		$96 \times 10/100$	$10/100$
	Leaky_ReLU19		N/A	N/A

$\mathbf{x}_b$ is a sequentical concatenation of $\mathbf{b}_0'$~$\mathbf{b}_{18}'$;
avg-pool: average pooling with window size of 1 and stride of 2;
pad: padding zero channels to match the quantity of out-channels for summation.

Table B.4: Parameters for ResNet20 (refer to table B.1).

Block (shortcut)	Original layer	Equivalent layer	Parameters	Out-channel feature maps
	Input ($\mathbf{x}_n$)	Input ($[\mathbf{x}_n; \mathbf{x}_b]$)	N/A	N/A
	Conv0 ($\mathbf{w}_0$)	Conv0 ($\mathbf{w}_0$)	N/A	N/A
	B0 (b_0)	B0 ($\mathbf{x}_b[b_0]$)	N/A	N/A
	ReLU0		N/A	N/A
Residual 0 (identity)	B1 (b_1)	B1 ($\mathbf{x}_b[b_1]$)	N/A	N/A
	Conv1 ($\mathbf{w}_1$)	Conv1 ($\mathbf{w}_1$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	B2 (b_2)	B2 ($\mathbf{x}_b[b_2]$)	N/A	N/A
	ReLU1		N/A	N/A
	B3 (b_3)	B3 ($\mathbf{x}_b[b_3]$)	N/A	N/A
	Conv2 ($\mathbf{w}_2$)	Conv2 ($\mathbf{w}_2^{'} = \mathbf{w}_2 \times M_2$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	M2 (M_2)			
	B4 (b_4)	B4 ($\mathbf{x}_b[b_4]$)	N/A	N/A
	ReLU2		N/A	N/A
Residual 1 (identity)	B5 (b_5)	B5 ($\mathbf{x}_b[b_5]$)	N/A	N/A
	Conv3 ($\mathbf{w}_3$)	Conv3 ($\mathbf{w}_3$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	B6 (b_6)	B6 ($\mathbf{x}_b[b_6]$)	N/A	N/A
	ReLU3		N/A	N/A
	B7 (b_7)	B7 ($\mathbf{x}_b[b_7]$)	N/A	N/A
	Conv4 ($\mathbf{w}_4$)	Conv4 ($\mathbf{w}_4^{'} = \mathbf{w}_4 \times M_4$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	M4 (M_4)			
	B8 (b_8)	B8 ($\mathbf{x}_b[b_8]$)	N/A	N/A
	ReLU4		N/A	N/A
Residual 2 (identity)	B9 (b_9)	B9 ($\mathbf{x}_b[b_9]$)	N/A	N/A
	Conv5 ($\mathbf{w}_5$)	Conv5 ($\mathbf{w}_5$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	B10 (b_{10})	B10 ($\mathbf{x}_b[b_{10}]$)	N/A	N/A
	ReLU5		N/A	N/A
	B11 (b_{11})	B11 ($\mathbf{x}_b[b_{11}]$)	N/A	N/A
	Conv6 ($\mathbf{w}_6$)	Conv6 ($\mathbf{w}_6^{'} = \mathbf{w}_6 \times M_6$)	$3 \times 3 \times 32 \times 32$	$32 \times 32 \times 32(32, 768)$
	M6 (M_6)			
	B12 (b_{12})	B12 ($\mathbf{x}_b[b_{12}]$)	N/A	N/A
	ReLU6		N/A	N/A
Residual 3 (avg-pool+pad)	B13 (b_{13})	B13 ($\mathbf{x}_b[b_{13}]$)	N/A	N/A
	Conv7 ($\mathbf{w}_7, s = 2$)	Conv7 ($\mathbf{w}_7, s = 2$)	$3 \times 3 \times 32 \times 64$	$16 \times 16 \times 64(16, 384)$
	B14 (b_{14})	B14 ($\mathbf{x}_b[b_{14}]$)	N/A	N/A
	ReLU7		N/A	N/A
	B15 (b_{15})	B15 ($\mathbf{x}_b[b_{15}]$)	N/A	N/A
	Conv8 ($\mathbf{w}_8$)	Conv8 ($\mathbf{w}_8^{'} = \mathbf{w}_8 \times M_8$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64(16, 384)$
	M8 (M_8)			
	B16 (b_{16})	B16 ($\mathbf{x}_b[b_{16}]$)	N/A	N/A
	ReLU8		N/A	N/A
Residual 4 (identity)	B17 (b_{17})	B17 ($\mathbf{x}_b[b_{17}]$)	N/A	N/A
	Conv9 ($\mathbf{w}_9$)	Conv9 ($\mathbf{w}_9$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64(16, 384)$
	B18 (b_{18})	B18 ($\mathbf{x}_b[b_{18}]$)	N/A	N/A
	ReLU9		N/A	N/A
	B19 (b_{19})	B19 ($\mathbf{x}_b[b_{19}]$)	N/A	N/A
	Conv10 ($\mathbf{w}_{10}$)	Conv10 ($\mathbf{w}_{10}^{'} = \mathbf{w}_{10} \times M_{10}$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64(16, 384)$
	M10 (M_{10})			
	B20 (b_{20})	B20 ($\mathbf{x}_b[b_{20}]$)	N/A	N/A
	ReLU10		N/A	N/A
Residual 5 (identity)	B21 (b_{21})	B21 ($\mathbf{x}_b[b_{21}]$)	N/A	N/A
	Conv11 ($\mathbf{w}_{11}$)	Conv11 ($\mathbf{w}_{11}$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64(16, 384)$
	B22 (b_{22})	B22 ($\mathbf{x}_b[b_{22}]$)	N/A	N/A
	ReLU11		N/A	N/A
	B23 (b_{23})	B23 ($\mathbf{x}_b[b_{23}]$)	N/A	N/A
	Conv12 ($\mathbf{w}_{12}$)	Conv12 ($\mathbf{w}_{12}^{'} = \mathbf{w}_{12} \times M_{12}$)	$3 \times 3 \times 64 \times 64$	$16 \times 16 \times 64(16, 384)$
	M12 (M_{12})			
	B24 (b_{24})	B24 ($\mathbf{x}_b[b_{24}]$)	N/A	N/A
	ReLU12		N/A	N/A
Residual 6 (avg-pool+pad)	B25 (b_{25})	B25 ($\mathbf{x}_b[b_{25}]$)	N/A	N/A
	Conv13 ($\mathbf{w}_{13}, s = 2$)	Conv13 ($\mathbf{w}_{13}, s = 2$)	$3 \times 3 \times 64 \times 96$	$8 \times 8 \times 96(6, 144)$
	B26 (b_{26})	B26 ($\mathbf{x}_b[b_{26}]$)	N/A	N/A
	ReLU13		N/A	N/A
	B27 (b_{27})	B27 ($\mathbf{x}_b[b_{27}]$)	N/A	N/A
	Conv14 ($\mathbf{w}_{14}$)	Conv14 ($\mathbf{w}_{14}^{'} = \mathbf{w}_{14} \times M_{14}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96(6, 144)$
	M14 (M_{14})			
	B28 (b_{28})	B28 ($\mathbf{x}_b[b_{28}]$)	N/A	N/A
	ReLU14		N/A	N/A
Residual 7 (identity)	B29 (b_{29})	B29 ($\mathbf{x}_b[b_{29}]$)	N/A	N/A
	Conv15 ($\mathbf{w}_{15}$)	Conv15 ($\mathbf{w}_{15}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96(6, 144)$
	B30 (b_{30})	B30 ($\mathbf{x}_b[b_{30}]$)	N/A	N/A
	ReLU15		N/A	N/A
	B31 (b_{31})	B31 ($\mathbf{x}_b[b_{31}]$)	N/A	N/A
	Conv16 ($\mathbf{w}_{16}$)	Conv16 ($\mathbf{w}_{16}^{'} = \mathbf{w}_{16} \times M_{16}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96(6, 144)$
	M16 (M_{16})			
	B32 (b_{32})	B32 ($\mathbf{x}_b[b_{32}]$)	N/A	N/A
	ReLU16		N/A	N/A
Residual 8 (identity)	B33 (b_{33})	B33 ($\mathbf{x}_b[b_{33}]$)	N/A	N/A
	Conv16 ($\mathbf{w}_{16}$)	Conv16 ($\mathbf{w}_{16}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96(6, 144)$
	B34 (b_{34})	B34 ($\mathbf{x}_b[b_{34}]$)	N/A	N/A
	ReLU17		N/A	N/A
	B35 (b_{35})	B35 ($\mathbf{x}_b[b_{35}]$)	N/A	N/A
	Conv18 ($\mathbf{w}_{18}$)	Conv18 ($\mathbf{w}_{18}^{'} = \mathbf{w}_{18} \times M_{18}$)	$3 \times 3 \times 96 \times 96$	$8 \times 8 \times 96(6, 144)$
	M18 (M_{18})			
	B36 (b_{36})	B36 ($\mathbf{x}_b[b_{36}]$)	N/A	N/A
	ReLU18		N/A	N/A
	g_p		N/A	N/A
	FC ($\mathbf{w}_{fc}$)		$96 \times 10/100$	$10/100$
	ReLU19		N/A	N/A

M_i denotes a multiplier [1];
$\mathbf{x}_b$ is a sequentical concatenation of b_0~b_{36}.

Table B.5: Parameters for ResNet20-Fixup (refer to table B.4).