1-9-2023

Normalization and Generalization in Deep Learning

Griffin Hurt
gxh2932@rit.edu

NORMALIZATION AND GENERALIZATION IN DEEP LEARNING

by

Griffin Hurt

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Applied and Computational Mathematics

School of Mathematical Sciences, College of Science

Rochester Institute of Technology

Rochester, NY

January 9, 2023

Committee Approval:

Nathan Cahill, D.Phil. School of Mathematical Sciences Thesis Advisor	Date
Ernest Fokoué, Ph.D. School of Mathematical Sciences Committee Member	Date
Mathew Hoffman, Ph.D. School of Mathematical Sciences Committee Member	Date
Kara Maki, Ph.D. School of Mathematical Sciences Director of M.S. Program	Date

Abstract

In this thesis, we discuss the importance of data normalization in deep learning and its relationship with generalization. Normalization is a staple of deep learning architectures and has been shown to improve the stability and generalizability of deep learning models, yet the reason why these normalization techniques work is still unknown and is an active area of research. Inspired by this uncertainty, we explore how different normalization techniques perform when employed in different deep learning architectures, while also exploring generalization and metrics associated with generalization in congruence with our investigation into normalization. The goal behind our experiments was to investigate if there exist any identifiable trends for the different normalization methods across an array of different training schemes with respect to the various metrics employed. We found that class similarity was seemingly the strongest predictor for train accuracy, test accuracy, and generalization ratio across all employed metrics. Overall, BatchNorm and EvoNormBO generally performed the best on measures of test and train accuracy, while InstanceNorm and Plain performed the worst.

Acknowledgements

I could not have done this without Dr. Christopher Kanan and especially not without Dr. Nathan Cahill. I cannot thank them enough. I'd also like to thank RIT's Research Computing Cluster for providing me with the resources to make the experiments in this thesis possible, as well as Dr. Ernest Fokoue and Dr. Matthew Hoffman for taking the time out of their lives to serve on the committee overseeing this thesis. Finally, I'd like to thank my loving family for supporting me through this process. Without them, none of this would have been possible.

Contents

Mathematical Preliminaries

Definition 1 (Perceptron)**.** A perceptron is defined as a single-layer network with outputs defined by

$$\mathbf{y}(\mathbf{x}) = \sigma\left(\mathbf{W}^{\mathbf{T}}\mathbf{x} + \mathbf{w}_0\right).$$

where $\mathbf{x} = [x_1, \ldots, x_d] \in \mathbb{R}^d$, $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_c] \in \mathbb{R}^{d\times c}$, $\mathbf{w}_0 = [w_{0,0}, \ldots, w_{c-1,0}]^{\mathbf{T}} \in \mathbb{R}^c$, $\mathbf{y}(\mathbf{x}) = [y_0(\mathbf{x}), \ldots, y_{c-1}(\mathbf{x})]^{\mathbf{T}}$ and σ is some activation function. The letter d refers to the number of features in the input and c refers to the number of classes in which the input can be classified.

Definition 2 (Multilayer perceptron)**.** A multilayer perceptron of depth N is defined as

$$\mathbf{y}(\mathbf{x}) = \sigma\left(\mathbf{W}^{(N)^{\mathbf{T}}} \cdots \sigma\left(\mathbf{W}^{(2)^{\mathbf{T}}}\sigma\left(\mathbf{W}^{(1)^{\mathrm{T}}}\mathbf{x} + \mathbf{w}_1\right) + \mathbf{w}_2\right) \cdots + \mathbf{w}_N\right).$$

Definition 3 (Loss function)**.** The loss function is defined as some function $L : \mathbb{R}^c \to \mathbb{R}$ which acts as the objective function of a model.

Definition 4 (Batch)**.** A batch is defined as a set $\mathcal{B} = \{\mathbf{x}_1 \ldots \mathbf{x}_n\}$, where m is the number of elements in the batch.

Introduction

Deep learning is a relatively new field that has been growing exponentially in recent years and has seen applications in many different areas, including self-driving vehicles, personal assistants, medical image analysis, and much more. There are three main types of deep learning: supervised, unsupervised, and reinforcement learning. Supervised learning is where the algorithm is given a set of training data, and the desired outputs are also known [1]. The algorithm then learns to map the input data to the desired output. Unsupervised learning is where the algorithm is given a set of data but not told what the desired output should be [1]. The algorithm then has to learn to find patterns and structure in the data. Reinforcement learning is where the algorithm is given a set of data and a reward function. The algorithm then learns to map the input data to the desired output in order to maximize the reward [1].

Within these three types exists a bimodal set of study: computer vision and natural language processing. Computer vision is the process of using computers to interpret and understand digital images [1]. Computer vision can be used for a variety of purposes, such as image recognition, object detection, and motion estimation. Natural language processing is concerned with language and symbolic representations [1]. Perhaps the quintessential example of natural language processing is language translation. This thesis is mainly concerned with computer vision using supervised learning.

Across all these fields of studying exists a common principle: generalization. Generalization is the ability of a model to accurately predict outputs for data that was not used in the training of the model [1]. The concept is fundamental to the field of deep learning, and on an even greater scale, artificial intelligence. The purpose of developing deep learning models is to be able to predict outcomes, and in some cases, act on those outcomes. Models make predictions based on things that they have already learned in training. To make proper predictions, they need to be able to generalize outside of just the data seen in training. That is what makes a "good" model. It's no wonder then why the concept of generalization is such an important and highly studied topic within deep learning. It's much easier to create models which generalize well when you know what properties of a model are associated with its ability to generalize.

Normalization is one of the techniques used to help deep learning models generalize well, and are a fundamental building block of most deep learning architectures [2, 3, 4, 5, 6]. The topic has been studied

fairly extensively and many different normalization techniques have been developed over the years as a result of these studies [7, 8, 9, 10, 11, 12, 13, 14, 15]. However, the question of *why* these normalization techniques help performance remains somewhat of a mystery.

In this thesis, we will review some of the different models and normalization techniques commonly used as well as an array of techniques for evaluating and investigating model performance. Once these have been reviewed, we conduct a collection of experiments in an attempt to uncover any potential trends between the different models, normalization techniques, and metrics. Our ultimate goal is to see if there exist any commonalities between the different normalization schemes with respect to our tested metrics, so that we may gain insight into what may or may not make a good scheme.

Chapter 1

Normalization

Normalization in deep learning traces its origins to Yann Lecunn et al. in the foundational paper *Efficient Backprop* [16], which laid the groundwork for much of deep learning theory today. In this paper, the authors highlight the idea that if you normalize your inputs to your model, convergence toward a solution will typically be faster. As important as this idea was, there was a lack of subsequent novelty for quite some time. That is until the introduction of the normalization technique known as BatchNorm from Ioffe and Szegedy [9]. Today, most architectures employ either BatchNorm or LayerNorm in the intermediate layers of the model, similar to Figure 1.1. BatchNorm tends to be more popular for vision-related architectures such as CNNs, while LayerNorm tends to be more popular for language-related architectures such as Transformers. As for why these normalization techniques work, it is still not totally clear and research into this topic is still ongoing. Slightly later in this chapter, we shed some light as to where and how research is being done to confront this issue.

1.1 Why Is It Important?

The paper which introduced Batch Normalization [9] is one of the most highly cited papers in the field of machine learning, netting over 35,000 citations to date. Most modern architectures employ some form of data normalization for training and testing and has become a fundamental building block for constructing models. From MLPs, to ResNets, to Transformers, you are almost guaranteed to spot some sort of normalization step within their structure. This almost universal prevalence makes understanding why normalization lets models operate more smoothly than without, and on a more general scale, why neural networks actually work, an important task. As an analogy, if we wish to understand the brain, we need to understand its parts.

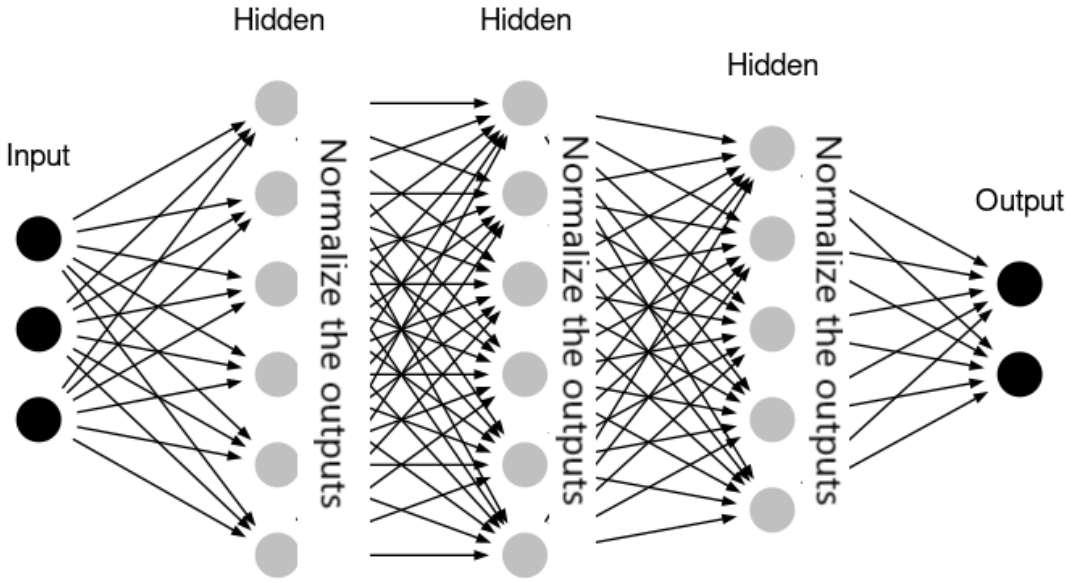


Figure 1.1: Multilayer perceptron with normalization [17].

1.2 Why It Works

The answer to this question does not have a clear answer. As stated earlier, the authors who published the paper on Batch Normalization believed that their method worked due to reducing what is known as Internal Covariate Shift [9], which is defined as the change of the distribution of the activations induced by the change in the network parameters during training. It was later shown that preventing internal covariance shift was not necessarily indicative of better training [18]. Instead, it was proposed that BatchNorm's effectiveness may have been connected to how it impacts the "Lipschitzness" of the loss function. The problem is still unsolved though, and there is still not a conclusive argument for why BatchNorm or other forms of normalization impact performance in the way that they do.

1.3 The Current State of Research

Probably the most interesting paper as of late was published by Sing et al. [6], in which they conducted a large study looking at different normalization techniques employed in different CNNs for image classification while sweeping over different hyperparameters, such as batch size and learning rate. The paper was mostly focused on how normalization impacts training behavior. Their key findings were i) non-weight-based normalization schemes were found to prevent feature explosion during forward propagation, ii) how well a normalizer generates dissimilar features between classes is a strong predictor of optimization speed, and iii) small group sizes for channel-based normalization techniques (GroupNorm, LayerNorm, Instan-

ceNorm) are connected to large gradient norms in earlier layers which can lead to gradient explosion and therefore unstable backpropagation. Although this paper does not really provide an argument for why normalization works in general, it may provide a good basis to work from for solving that problem.

1.4 BatchNorm

Input: Values of $\mathbf{x}$ over a mini-batch of size n: $\mathcal{B} = \{\mathbf{x}_{1\ldots n}\}$;
Parameters to be learned: γ, β
Output: $\{\mathbf{y}_i = \text{BN}_{\gamma,\beta}(\mathbf{x}_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$\mathbf{y}_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(\mathbf{x}_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to input $\mathbf{x}$ over a mini-batch [9]. Informally, we can think of the algorithm as normalizing each feature of an input relative to the other inputs within the mini-batch, and then scaling and shifting the normalized inputs.

Following early theory surrounding the normalization of inputs [16], Ioffe and Szegedy took this principle and used it to formulate what is known as BatchNorm [9]. They theorized that the distribution of the inputs changed as they passed through the model due to the change in network parameters during training. They coined this change in distribution as "internal covariate shift" and hypothesized that this internal covariate shift contributed to greater model sensitivity to learning rates and required careful initialization of the parameters. They believed that the solution to their theorized issue was Batch Normalization. Their empirical results which implemented Batch Normalization within CNNs with the purpose of image classification were shown to be a massive improvement over the previous state-of-the-art architectures, which did not possess intermediary normalization layers.

1.4.1 Internal Covariate Shift

Internal Covariate Shift is defined as the change of the distribution of the activations induced by the change in the network parameters during training [9]. Ioffe and Szegedy believed that if they could correct this covariate shift then this would improve training. They believed that by fixing the distribution of the inputs at mean 0 and variance 1 throughout the network that this would prevent internal covariate shift, and by preventing the internal covariate shift, improve training speed and stability. This principle was inspired by

two papers [16, 19], which showed that network training converges faster if its initial inputs are linearly transformed to have mean 0 and variance 1.

While internal covariance shift coincided with improved training in the empirical results, it was later shown that preventing internal covariance shift was not necessarily indicative of better training [18]. They showed that explicitly introducing covariate shift into a model using BatchNorm did not lead to worse training outcomes. Following this, the authors proposed that instead of looking at internal covariance shift as the main metric of insight, we should instead perhaps look towards the parameter space, and how BatchNorm impacts the concavity of the parameter space, and hence, the smoothness of the loss surface.

1.4.2 Smoothness

Santurkar et al. showed that the implementation of BatchNorm led to a smoother loss surface by looking at the "Lipschitzness" of the loss function [18]. Their argument details that the smoother the loss surface, the more informed the gradient traversal of the surface will be. This is because the gradient is a first-order method and so it has no knowledge of curvature, and so the greater the curvature of the surface the less accurate the gradient traversal will be since it cannot account for any present curvature in the direction it wishes to travel in as informed by the gradient. Hence, the surface smoothing introduced by BatchNorm will generally lead to better training outcomes for training schemes that employ gradient descent.

Looking at the concept of "Lipschitzness", let us first explain what it is. In essence, Lipschitzness refers to the magnitude of the gradient of the loss function with respect to the activations or $||\nabla_{y_j} \mathcal{L}||$. The term itself stems from the concept of Lipschitz continuity and the Lipschitz constant. The reason why the Lipschitzness is important is because it dictates the change in the loss when taking a step with gradient descent. The Lipschitz constant of a function gives a measure of how much the output of the function can change for a given change in the input. A smaller Lipschitz constant indicates that the function is more smooth and has a more predictable gradient, while a larger Lipschitz constant indicates that the function is less smooth and has a less predictable gradient.

The authors show that the gradient magnitude is bounded for a batch normalized network and can achieve "better" Lipschitzness compared to the non-normalized network; "better" in this case means that the Lipschitz constant indicates a more predictive gradient since it is smaller. The authors also show that the quadratic form of the Hessian matrix of the loss function with respect to the activations for models using layer-wise BatchNorm is upper-bounded, indicating a limit to the "unsmoothness" of the loss surface, and so the gradient is generally more predictive compared to a model that does not employ layer-wise BatchNorm.

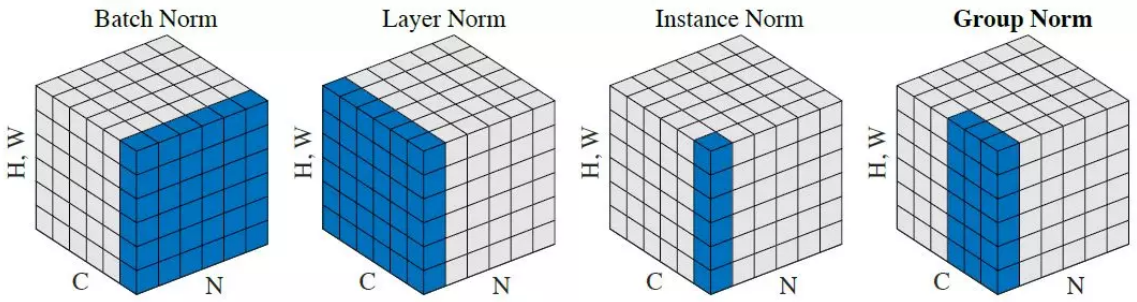


Figure 1.2: Feature tensors with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes [11].

1.5 Normalization Before BatchNorm

Prior to the advent of BatchNorm and its massive success, there existed normalization techniques that were implemented in CNNs: Local Contrast Normalization [7] and Local Response Normalization [8]. However, these were fairly niche compared to BatchNorm since they only really saw usage in CNNs and not so much in the broader sphere of deep learning models, since they were designed for tasks related to computer vision. Following the introduction of BatchNorm and other regularization techniques though, these methods mostly fell out of favor and are generally no longer used as a result of being rendered obsolete.

1.6 BatchNorm's Descendants

Following BatchNorm and its success, many took the opportunity to build upon it and develop different normalization techniques with different purposes and different results. Perhaps the most notable of these descendants is LayerNorm [10]. Today, LayerNorm is considered the state-of-the-art normalization technique in the field of natural language processing. The famous Transformer architecture which has taken the deep learning scene by storm makes use of LayerNorm [5]. Interestingly, the computer vision variant of the Transformer architecture [4] also employs LayerNorm just like its ancestor, which is a deviation from the norm of vision-related architectures which typically employ BatchNorm [3, 2, 20, 21, 22, 23, 24, 25]. Outside of LayerNorm, other techniques include GroupNorm [11], Instance Normalization [12], Variance Normalization [26] and many more. [1]

LayerNorm LayerNorm was developed with the intention of bringing the revolution of normalization to recurrent neural networks [10]. LayerNorm is actually quite similar to BatchNorm, except that the way it normalizes its data is by normalizing a different dimension of the activations. Instead of normalizing by the batch statistics, LayerNorm normalizes by the neuron statistics. In other words, for LayerNorm, the mean and variance statistics are calculated across the feature dimension, for each element and instance

[1] The operations performed by the normalization procedures described in this section can be found in the table.

Activations-Based Normalizers $\mu_{\{d\}} = \mu_{\{d\}}(\mathcal{A}); \sigma_{\{d\}} = \sigma_{\{d\}}(\mathcal{A})$	
BN [9]	$\frac{\mathcal{A}-\mu_{\{b,x\}}}{\sigma_{\{b,x\}}}$
LN [10]	$\frac{\mathcal{A}-\mu_{\{c,x\}}}{\sigma_{\{c,x\}}}$
IN [12]	$\frac{\mathcal{A}-\mu_{\{x\}}}{\sigma_{\{x\}}}$
GN [11]	$\frac{\mathcal{A}-\mu_{\{c/g,x\}}}{\sigma_{\{c/g,x\}}}$
FRN [13]	$\frac{\mathcal{A}}{\overline{\text{RMS}}_{\{x\}}}$
VN [14]	$\frac{\mathcal{A}}{\sigma_{\{b,x\}}}$
ENBO [15]	$\frac{\mathcal{A}}{\max\{\sigma_{\{b,x\}}, v\odot\mathcal{A}+\sigma_{\{x\}}\}}$
ENSO [15]	$\frac{\mathcal{A}\rho(v\odot\mathcal{A})}{\sigma_{\{c/g,x\}}}$

Figure 1.3: Operations performed by different normalizers. $\mathcal{A}$ denotes normalization layer input (activations or activation function output). Operators $\mu_{\{d\}}(\mathcal{A})$ and $\sigma_{\{d\}}(\mathcal{A})$ calculate the mean and standard deviation along the dimensions specified by a set $\{d\}$. Symbols b, c, x denote the batch, channel, and spatial dimensions. The notation c/g denotes division of c neurons (or channels) into groups of size g. Each group is normalized independently when grouping is performed. RMS denotes root mean square.

independently. For BatchNorm, the statistics used for normalization are calculated across all elements of all instances in a batch.

Instance Normalization Instance Normalization was developed with the intention of developing a more efficient normalization technique than BatchNorm for the process of image generation [12]. Instance Normalization can be thought of in terms of BatchNorm, where instead of normalizing using the entire batch, you normalize by each sample in the batch (i.e. only one blue column in the BatchNorm cube from Figure 1.2). Instance Normalization has been known to be fairly unstable during training [6]. This instability is believed to be related to gradient explosion during training which leads to unstable backpropagation. The reason for this is theorized to be related to the limited scope of how InstanceNorm normalizes the data. Unlike other norms which normalize by groups of channels or by batches, InstanceNorm normalizes by each channel for each individual sample in the batch as seen in Figure 1.2.

Group Normalization Group Normalization, like LayerNorm and InstanceNorm, is independent of batch size [11]. GroupNorm computes normalization statistics by grouping the channels and normalizing within the groups. You can actually think of LayerNorm and InstanceNorm in terms of GroupNorm, where LayerNorm is GroupNorm with group size C and InstanceNorm is GroupNorm with group size 1. GroupNorm was created with the intention of providing an alternative normalization technique to BatchNorm such that it would be a more stable technique that was not reliant on batch statistics, since it had been observed that the error of models employing BatchNorm would witness rapidly increasing error when the batch size was reduced [11], because the batch statistics were then less representative of the greater set of inputs. Generally, larger models need smaller batch sizes because of computational limitations with respect to memory, and

so using BatchNorm on large models could potentially be less fruitful than on smaller models. GroupNorm was designed such that this trade-off between batch size and memory would not be an issue.

Filter Response Normalization Filter Response Normalization follows the footsteps of GroupNorm in trying to develop a normalization technique that is independent of the batch [13]. It works by normalizing the inputs per channel and then sending those normalized inputs through an activation function known as the Threshold Linear Unit (TLU). The reason for the activation function is that Filter Response Normalization does not have mean centering, which leads to the activations typically having some non-zero mean, and this non-zero mean does not interact well with the ReLU activation function and can lead to poor performance. In essence, TLU is ReLU with an additional learned parameter τ where ReLU is defined as $\max(y, 0)$ and TLU is defined as

$$\max(y, \tau) = \max(y - \tau, 0) + \tau = \text{ReLU}(y - \tau) + \tau.$$

As the name might suggest, Filter Response Normalization is quite similar to the previously mentioned Layer Response Normalization. They do share some differences, however; the most apparent being that Layer Response Normalization operates on adjacent channels at the same spatial location, while FRN operates globally over the spatial extent.

EvoNorm EvoNorms are a result of experiments with the purpose of finding optimal normalization techniques through an automated approach [15]. These norms exist as a unification of normalization and activation into a single computation graph. There exist two different forms of EvoNorms: B-series and S-series. B-series EvoNorms are batch-dependent while S-series are batch independent and operate on individual samples.

Variance Normalization Variance Normalization is simply an ablation of BatchNorm which does not employ mean centering [14]. It was designed for experiments investigating the rank-preserving properties of BatchNorm. It has been shown that mean centering is redundant for linear networks, but the authors of Variance Normalization show that empirically this may also be true for non-linear networks.

Parametric Normalization Another type of normalization employed in deep learning architectures is parametric normalization, as opposed to activation normalizers. Parametric normalizers such as Weight Normalization [26] operate in the weight space as opposed to the activation space. Meaning, that instead of normalizing the activations, parametric methods normalize the weights. In this paper, however, we do not inspect parametric normalization and leave that for future work.

On a more general note, typically once activations are normalized, they will then be scaled and shifted using variables γ and β respectively, where γ and β are learned affine parameters. An example of this can be seen in Algorithm 1. Once they are scaled and shifted, they are typically passed through some non-linearity such as ReLU. This is not always the case though, as can be seen with the Transformer which does not pass the normalized activations through an activation function (unless the normalizer architecture includes one, such as Filter Response Normalization with TLU).

Chapter 2

Generalization & Flatness

2.1 Generalization

One of, if not the most important goals in deep learning, is training networks that can generalize well. Why, and how models generalize is still a subject of great interest, yet with inconclusive results. There has been a myriad of papers [27, 28, 29, 30**?** , 31, 32, 33] which have tackled aspects of the issue from a theoretical perspective, but still much mystery remains. The two main trajectories traveled for investigating the generalization problem are typically either from a PAC-Bayes perspective or from a geometrical analysis of the loss surface.

The PAC-Bayes approach typically involves generating some type of bound on how well a model will generalize based on how it was trained. The geometric approach focuses on metrics of sharpness/flatness of minima. There has been an observed phenomenon in which flat minima tend to generalize better than their sharp counterparts. There has been a plethora of research backing this trend and remains quite a promising avenue [34, 35, 36, 37, 38, 39, 40]. Still, the hunt for why networks generalize and how to predict how well they will generalize ensues. We are interested in contributing to this search by studying the relationship between normalizers and how they may influence the geometry of the loss surface, as well as how they impact generalization. For the sake of time, we will not be venturing into the realm of PAC-Bayes analysis and leave that for future work. Instead, we will be directing our focus toward metrics of flatness as it pertains to the loss surface.

Generalization is at the core of artificial intelligence research. Being able to create an agent which can effectively generalize what it has learned is arguably the most important goal in the entire field. Hence why research into this area is so extensive. The term "generalize" in deep learning is used fairly loosely. Usually, the data that's used to test how well a model generalizes is *very* similar to the data that it was trained on, at least in the case of supervised learning. The train and test sets for things like ImageNet [41] or CIFAR [42]

both have images of cats and dogs, for example. It is not as if the test set has classes of objects not seen in the train set. The space in which the data lies is very limited. So when we mention the term "generalize" we are just referring to the model's ability to evaluate data from the test set.

2.2 Flatness

2.2.1 What is a Loss Surface?

A loss surface is a surface that represents the error of a model as a function of its weights. The loss surface is generally used to geometrically schematize how the error changes as the weights of a neural network are updated during training. Visualizing loss surfaces is not a trivial problem since most models are high-dimensional (i.e. have many parameters). Probably the most popular approach to visualizing loss surfaces was done by Li et al. [38].

In their paper, they outline two different approaches to tackling the problem. The first approach involves 1-dimensional linear interpolation. To do this, two sets of parameters θ and θ' are chosen, and then values of the loss function along the line connecting these two points are plotted. This line can be parameterized by choosing a scalar parameter α, and defining the weighted average $\theta(\alpha) = (1 - \alpha)\theta + \alpha\theta'$. The function $f(\alpha) = L(\theta(\alpha))$ is then plotted.

The second approach involves generating contour plots by randomly sampling direction vectors. This approach involves choosing a center point θ^* in the graph and then choosing two direction vectors: δ and β. Then a contour plot is generated by plotting the function $f(\alpha) = L(\theta^* + \alpha\delta)$ in the 1-D case or $f(\alpha, \beta) = L(\theta^* + \alpha\delta + \beta\eta)$ in the 2-D case.

Another method, outlined in [43], uses the eigenvectors of the Hessian of the loss function. To plot the loss surface, they first compute the top Hessian eigenvector and then perturb the model parameters along that direction and measure the loss. The loss value is then plotted as a function of the perturbation. They also introduce another very similar method which uses the gradient instead of the eigenvector.

2.2.2 Relationship Between Flatness and Generalization

There exist a series of notable works which link flatness of the loss surface to generalization. The idea that flat minima tend to generalize well was first introduced in 1997 by Hochreiter and Schmidhuber [34] in which they argue that flat minima correspond to low expected generalization error. This idea didn't see much more attention until the paper published by Keskar et al. [35], who showed that large batch sizes during training can yield sharper minima than smaller batch sizes ($32 - 512$), and that sharp minima generalize poorly. This trend was subsequently observed in several other influential papers [36, 37, 38, 39, 40].

However, it has also been shown that flatness is not necessarily indicative of how well a model generalizes [44]. Through reparameterization of the model such that the area surrounding the minimum found by the model is sharpened, the model can still generalize just as well. While this result may prevent any affirmative claims about how well a model will generalize just by inspecting flatness, it can still be a valuable signal.

At its core, the flatness of the loss surface is a metric for measuring how sensitive a model is to parameter perturbations. If small perturbations to the model parameters result in large increases in error then the minimum is sharp. More generally, a small step from the optimal set of parameters results in a large change in the loss function output (relatively speaking). Sharp minima tend to possess rigid class boundaries [45], hence why small perturbations to the parameters of the model can have large impacts on these class boundaries which result in misclassification. Flat minima, however, tend to have less rigid, wider, nicer boundaries. These more lenient boundaries can allow the model to better classify data not seen in the training set (i.e. generalize better), since a classifier whose decision boundaries exist far from the original training data will be more likely to classify test data correctly than otherwise.

2.3 Hessian-Based Analysis

The Hessian of the loss function with respect to the model parameters can provide insight into the second-order behavior of the model. This is of particular interest to us given the observations between the curvature of the loss surface and a model's ability to generalize.

The Hessian matrix is a square matrix that is used to describe the second-order derivatives of a multivariate function. More formally, suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a function taking as input a vector $x \in \mathbb{R}^n$ and outputting a scalar $f(x) \in \mathbb{R}$. If all second-order partial derivatives of f exist, then the Hessian matrix H of f is a square $n \times n$ matrix, usually defined and arranged as follows:

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

The eigenvalues of the Hessian matrix are used to describe the curvature of the function. If all of the eigenvalues are positive, then the function is said to be convex. If all of the eigenvalues are negative, then the function is said to be concave. If some of the eigenvalues are positive and some are negative, then the function is said to be saddle-shaped. If the function is flatter (closer to being linear), then it will have smaller eigenvalues, and if the eigenvalues are large in magnitude, then the function is far from being linear (curved).

In our analysis of the second-order behavior later in this thesis, we will consider two metrics: the absolute trace of the Hessian and the eigenvalue spectral density of the Hessian. The Hessian will be calculated with respect to the training set, since we are interested in where the model converges during training. If we calculated the Hessian with respect to the test set, then we would almost certainly be evaluating the curvature of the loss surface at a point somewhere different than where we converged, since the loss value for the test set is likely going to be different (and larger) than that of the training set.

2.3.1 Trace

Input: Parameter: θ.
Compute the gradient of θ by backpropagation, i.e., compute $g_\theta = \frac{dL}{d\theta}$.
Output: $Tr_{abs}(H) = \text{sum}\{\text{abs}\{\mathbb{E}[v \odot Hv]\}\}$
for $i = 1, 2, ...$ **do**
Draw a random vector v from Rademacher distribution (same dimension as θ).
Compute $gv = g_\theta^T v$
Compute Hv by backpropagation, $Hv = \frac{d(gv)}{d\theta}$
Compute and record $v \odot Hv$
end for
Compute the average of all computed $v \odot Hv$
Return the sum of the absolute-valued elements of the computed average

Algorithm 2: Absolute Trace computation.

A commonly employed metric for measuring curvature of a surface at a point is the trace of the Hessian evaluated at that point. The trace of a matrix is the sum of its diagonal entries. We employ a modified version of this metric by calculating the trace using the absolute value of each element. This is done to get a full picture of the curvature of the surface, since computing the trace the standard way will encounter issues when there exists saddling on the surface. Clearly, a flat (constant surface) would have a zero matrix as the Hessian, so the trace of the Hessian would be zero. However, consider the 2-D function $f(x, y) = x^2$–y^2. The origin is a saddle point – there is strong positive curvature in the x direction and strong negative curvature in the y direction, but at the origin, the trace of the Hessian is also zero. And furthermore, as you move away from the origin along the line $x = y$, the trace of the Hessian is zero for all points on that line. This effect may generally be less powerful with high dimensional functions such as deep nets due to having many parameters and hence unlikely to have "cancellations" to the same degree as the example, and also because stochastic gradient descent for deep networks tends to settle in flat regions rather than saddle-like regions, but taking the absolute trace still allows for additional precision without much extra compute.

Since computing the Hessian for models with many parameters is computationally infeasible, we instead approximate the diagonal of the Hessian using a modified form of the Hutchinson method [46], and then

compute the absolute trace using the approximated diagonal. To approximate the diagonal, we compute

$$Diag(H) \approx \mathbb{E}[v \odot Hv]$$

where v is a random vector drawn from a Rademacher distribution [47] (discrete distribution taking on only two possible values ± 1 with equal probability) such that $\mathbb{E}[vv^T] = I$, and $\odot$ denotes the Hadamard product (element-wise product). We compute Hv using the following observation

$$\frac{\partial g_\theta^T v}{\partial \theta} = \frac{\partial g_\theta^T}{\partial \theta} v + g_\theta^T \frac{\partial v}{\partial \theta} = \frac{\partial g_\theta^T}{\partial \theta} v = Hv \tag{2.1}$$

where

$$g_\theta = \frac{\partial L}{\partial \theta} \in \mathbb{R}^m$$

$$H = \frac{\partial^2 L}{\partial \theta^2} = \frac{\partial g_\theta}{\partial \theta} \in \mathbb{R}^{m \times m}$$

and L denotes the loss function. In Equation (2.1), the first equality is the chain rule, the second is due to the independence of v with respect to θ, and the third equality is the definition of the Hessian. Note that the computational cost to find Hv is the same as one iteration of backpropagation, and hence much less expensive than computing the full Hessian. Once we've calculated Hv we then use it to calculate the diagonal of H as follows

$$\begin{aligned} \text{Diag}(H) = \text{Diag}(HI) = \text{Diag}\left(H\mathbb{E}\left[vv^T\right]\right) = \mathbb{E}\left[\text{Diag}\left(Hvv^T\right)\right] \\ = \mathbb{E}[v \odot Hv]. \end{aligned}$$

We then calculate the absolute trace by summing the absolute value of each element in the calculated diagonal. Once the trace has been computed, we normalize the trace by the number of parameters in the model. This is because we use models with different parameter counts, and models with more parameters will generally have a larger trace since their Hessian will be greater in dimension.

2.3.2 Eigenvalue Spectral Density

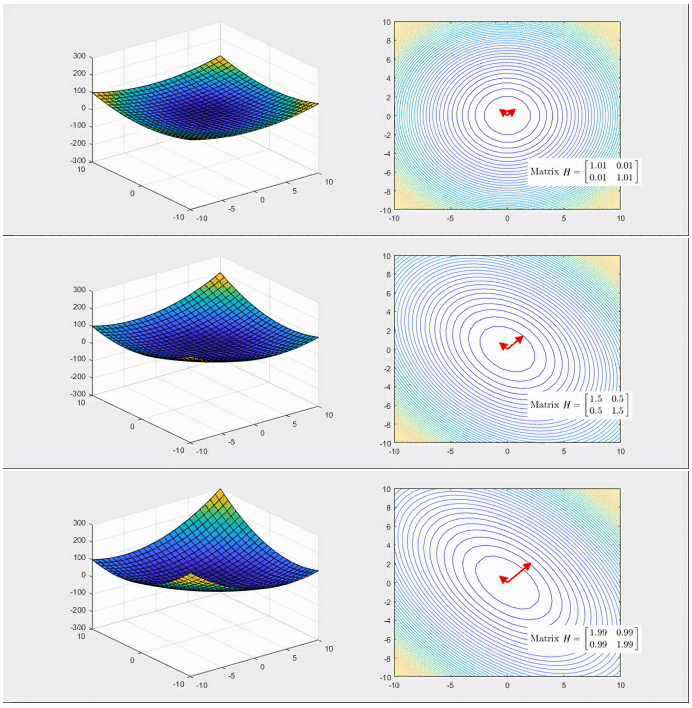


Figure 2.1: A surface centered about the origin and it's corresponding Hessian matrix H. The red vectors correspond to the eigenvectors of the Hessian.

We are interested in the eigenvalues of the Hessian matrix since they provide a window to peer into the geometry of the loss landscape, as can be seen in Figure 2.1. A Hessian with zero-valued eigenvalues denotes a flat surface. We operate in real-valued space and the Hessian matrix is both square and symmetric, and so has real-valued eigenvalues. This allows for a convenient and intuitive analysis of the loss surface. The eigenvectors of the Hessian correspond to the principal axes of a surface at a given point and their eigenvalues provide information on their magnitude. The greater the magnitude, the greater the curvature along the surface in the direction of the eigenvector from the point, and the lesser the magnitude the lesser the curvature.

We can think of the eigenvalue spectral density of a matrix as a probability density distribution that measures the likelihood of finding eigenvalues near some point on the real line. To compute the eigenvalue spectral density, we follow the procedure outlined by Yao et al. [43] known as stochastic Lanczos quadrature.

In summary, to approximate the spectral density of the Hessian, we apply the Lanczos algorithm (not to be confused with stochastic Lanczos quadrature) with q steps on the Hessian to get the tridiagonal matrix T of size $(q \times q)$. We then calculate the q eigenpairs of T which are used in a Guassian kernel f to approximate the distribution of eigenvalues for the Hessian. This method of calculating the eigenpairs for T and using them to approximate the eigenvalues of the Hessian is less expensive than directly calculating the Hessian's

eigenvalues. The full spectral density of the Hessian eigenvalues is defined as

$$\phi(t) = \frac{1}{m} \sum_{i=1}^{m} \delta\left(t - \lambda_i\right),$$

where δ is the Dirac distribution and λ_i is the i^{th} eigenvalue of the Hessian, H. To approximate $\phi(t)$, we apply a Guassian kernel f, with variance σ^2 to obtain

$$\phi_\sigma(t) = \frac{1}{m} \sum_{i=1}^{m} f(\lambda; t, \sigma),$$

where $f(\lambda; t, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-(t-\lambda)^2 / \left(2\sigma^2\right)\right)$. Note that

$$\mathrm{Tr}(f(H)) = \mathrm{Tr}\left(Qf(\Lambda)Q^T\right) = \mathrm{Tr}(f(\Lambda)),$$

where $Q\Lambda Q^T$ is the eigendecomposition of H, and $f(H)$ is the matrix function defined as

$$f(H) \triangleq Qf(\Lambda)Q^T \triangleq Q\,\mathrm{diag}\left(f\left(\lambda_1\right), \ldots, f\left(\lambda_m\right)\right) Q^T.$$

We approximate the Guassian kernel by

$$\phi_\sigma(t) = \frac{1}{m} \mathrm{Tr}(f(H; t, \sigma)),$$

which is computed using the Hutchinson method. This is done by drawing a random Rademacher vector v and computer the expectation $\mathbb{E}\left[v^T f(H; t, \sigma) v\right]$ to get

$$\phi_\sigma(t) = \frac{1}{m} \mathbb{E}\left[v^T f(H; t, \sigma) v\right].$$

This approximation is still infeasible however, since the trace would need to be computed for every value of t. Another approximation is made where we define $\phi_\sigma^v(t) = v^T f(H; t, \sigma) v$, which results in

$$\begin{aligned} \phi_\sigma^v(t) &= v^T f(H; t) v = v^T Q f(\Lambda; t) Q^T v \\ &= \sum_{i=1}^{m} \mu_i^2 f\left(\lambda_i; t\right), \end{aligned}$$

where μ_i is the dot product of v along the i^{th} eigenvector of H. We then approximate $\phi_\sigma^v(t)$ using the Riemann-Stieltjes integral

$$\phi_\sigma^v(t) = \int_{\lambda_m}^{\lambda_1} f(\alpha; t) d\pi(\alpha),$$

where the measure $\pi(\alpha)$ is defined as the piecewise function

$$\pi(\alpha) = \begin{cases} 0 & \alpha \leq \lambda_m \\ \sum_{i=1}^{j} \mu_i^2 & \lambda_j \leq \alpha \leq \lambda_{j-1} \\ \sum_{i=1}^{m} \mu_i^2 & \lambda_1 \leq \alpha. \end{cases}.$$

This integral is then approximated using Gaussian quadrature:

$$\phi_\sigma^v(t) \approx \sum_{i=1}^{q} \omega_i f\left(t_i; t, \sigma\right),$$

where (ω_i, t_i) is the weight-node pair. Stochastic Lancsoz is then used to approximate the quadrature, using q-steps. In this case, for SLQ we have q eigenpairs $\left(\tilde{\lambda}_i, \tilde{v}_i\right)$. With these eigenpairs, we perform the following approximation

$$\phi_\sigma^v(t) \approx \sum_{i=1}^{q} \omega_i f\left(t_i; t, \sigma\right) \approx \sum_{i=1}^{q} \tau_i f\left(\tilde{\lambda}_i; t, \sigma\right),$$

where $\tau_i = \left(\tilde{v}_i[1]\right)^2$, and $\tilde{v}_i[1]$ is the first component of $\tilde{v}_i$, . Finally, $\phi_\sigma(t)$ can be approximated using n_v runs of the Lanczos algorithm

$$\phi_\sigma(t) = \mathrm{Tr}(f(H)) \approx \frac{1}{n_v} \sum_{l=1}^{n_v} \left(\sum_{i=1}^{q} \tau_i^{(l)} f\left(\bar{\lambda}_i^{(l)}; t, \sigma\right) \right).$$

For additional detail behind stochastic Lanczos quadrature and how it is used to approximate eigenvalues, refer to the original paper [43]. For additional info on the Lanczos algorithm, Ch. 7 of *Applied Numerical Linear Algebra* by James Demmel [48] has a great explanation of how it works.

2.4 When Flatness Does and Does Not Work

Two very informative papers were recently published which outline the failure and success modes of measures of flatness for predicting generalization. The first paper by Zhang et al. [49] mostly focused on the sharpness measure developed by Keskar et al. [35] as well as Hessian-based eigenvalue measures. The second focuses on the maximum Hessian eigenvalue [50].

Zhang et al. [49] showed that for a series of classic image classification tasks (MNIST [51] and CIFAR-10 [42]), flatness measures change substantially as a function of epochs. This was shown by training for an additional 1000 epochs after reaching zero training error and observing a reduction in the flatness measure (more flat) even though the training error was no longer changing. Additionally, it is known that parameter re-scaling can arbitrarily change flatness [44], but it quickly recovers to a more typical value under further

training. It was also demonstrated that some variants of SGD (e.g. Adam optimizer) result in a significantly lesser correlation between flatness and generalization than found for vanilla SGD.

Kuar et al. [50] found that while larger learning rates reduce the maximum eigenvalue of the Hessian, the benefits of a larger learning rate disappear for large batch sizes. They also found that by scaling the batch size and learning rate together, it is possible to change $\lambda_{\max}$ without affecting generalization. The paper also discusses the effects of different types of regularization on $\lambda_{\max}$, namely sharpness-aware minimization (SAM). They found that while SAM produces smaller $\lambda_{\max}$ for all batch sizes, the benefits of SAM vanish with larger batch sizes. They also found that for dropout, excessively high dropout [1] probabilities can degrade generalization, even as they promote smaller $\lambda_{\max}$. Finally, they found that while batch-normalization does not consistently produce smaller $\lambda_{\max}$, it nevertheless confers generalization benefits.

[1] Dropout is a layer that randomly ignores features as they are passed through the layer.

Chapter 3

Vision Models

3.1 Convolutional Neural Networks

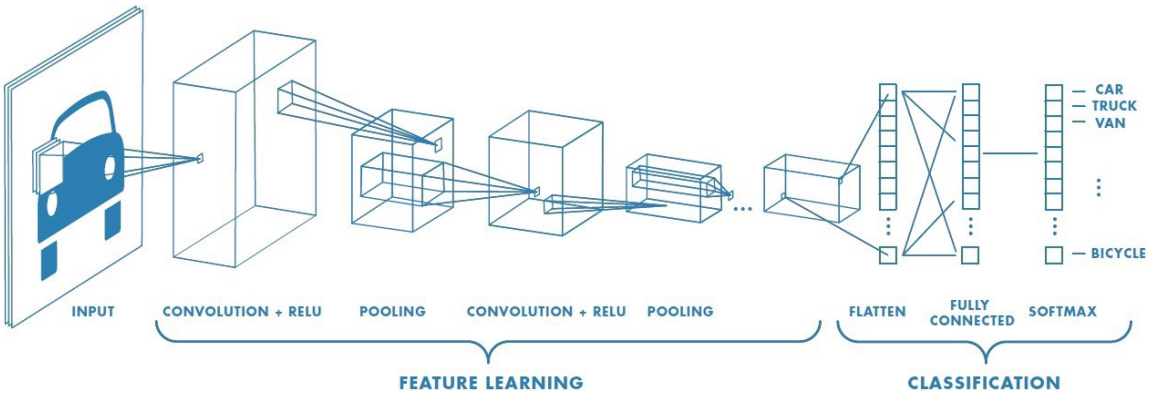


Figure 3.1: Basic diagram of CNN architecture [52].

The previous decade of computer vision and the successes that came with it can in large part be attributed to the effectiveness of convolutional neural networks (CNNs or ConvNets) as seen in Figure 3.1. The concept of scale in deep vision was transformed with the introduction of ConvNets and allowed for a pound-for-pound more powerful architecture than the classical multilayer perceptron. Perceptrons do not scale well with image size, in fact, the number of weights scales quadratically with square images (equal height and width). If we took an image of size $32 \times 32 \times 3$ (as seen in CIFAR-10/CIFAR-100 [42]), a fully-connected neuron in the first hidden layer of an MLP would have $32 \times 32 \times 3 = 3072$ weights. Now say we took a larger image of size $200 \times 200 \times 3$, then we would have $200 \times 200 \times 3 = 120,000$ weights. That is just one neuron and clearly, we would want more than just one if possible. ConvNets help alleviate this scaling issue. Unlike MLPs, ConvNets have layers with volumetric neurons arranged in 3 dimensions: width, height, color. For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions

$32 \times 32 \times 3$ (width, height, color respectively).

CNNs are typically comprised of five major building blocks (not counting normalization layers): the input, convolutional layers, ReLU activation functions, pooling layers, and a fully-connected layer at the end of the model. CNNs were designed with image input in mind and so the dimension of the input will typically take on the form of the number of inputs $\times$ height $\times$ width $\times$ channel (or color). The convolutional layers compute the output of neurons that are connected to local regions in the input. This is done by computing dot products between what are known as filters and the local feature regions in the input. The ReLU activation function simply takes in the input x and outputs $\max(0, x)$. Pooling layers perform what is known as downsampling along the spatial regions (height and width), in which they downsize the input so that there are fewer features. For example, you could have a downsampling of a $32 \times 32 \times 3$ input such that the resulting output of the downsampling is of size $16 \times 16 \times 3$. Finally, the fully-connected layer takes in the generated image features as inputs and computes the class score.

To give some more detail on how exactly the unique components (convolution and pooling) of a CNN operate, let us first look at what convolution actually is in this context. In essence, convolution acts as a way to help extract features from an input image and to reduce the number of parameters that need to be learned in order to improve the performance of the model. A convolutional layer accepts an input volume of size $H_1 \times W_1 \times C_1$ and requires four hyperparameters: the number of filters K, their spatial extent F, the stride S, and the amount of zero padding P. The convolutional layer produces an output volume of size $H_2 \times W_2 \times C_2$, where $H_2 = (H_1 - F + 2P)/S + 1$ and $W_2 = (W_1 - F + 2P)/S + 1$. In the output volume, the c-th channel slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the c-th filter over the input volume with a stride of S, and then offset by c-th bias.

Let us look at a concrete example to help convey the process. Consider an input volume of size $W_1 = 5$, $H_1 = 5$, $D_1 = 3$ and parameters $K = 2$, $F = 3$, $S = 2$, $P = 1$. Suppose our input is the following tensor:

$$\left[\begin{bmatrix} 2 & 2 & 2 & 2 & 0 \\ 2 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 2 & 2 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 2 & 1 \\ 2 & 1 & 0 & 2 & 1 \\ 2 & 1 & 2 & 1 & 0 \\ 2 & 0 & 0 & 2 & 2 \\ 0 & 2 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 2 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 2 & 2 & 2 \end{bmatrix}\right],$$

where each matrix corresponds to a channel. First, we apply the padding to our input:

$$\left[\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 & 2 & 0 & 0 \\ 0 & 2 & 2 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 2 & 2 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 1 & 2 & 1 & 0 \\ 0 & 2 & 1 & 0 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right].$$

Next, we take our filter of size $3 \times 3 \times 3$

$$\left[\begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix}\right],$$

with bias $= 1$, and our filter of size $3 \times 3 \times 3$

$$\left[\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & -1 & -1 \end{bmatrix}\right],$$

with bias $= 0$, and slide them over our input to get the output

$$\left[\begin{bmatrix} 6 & -3 & -1 \\ 7 & -5 & 7 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 10 & 12 & 4 \\ 13 & 11 & 8 \\ 6 & 11 & 1 \end{bmatrix}\right].$$

The output matrices are obtained by element-wise multiplication between each input matrix with its corresponding filter, summing it up, and then offsetting the sum by the bias. For example, let us look at how we would calculate the first entry in the first output matrix. Let $\star$ represent the operation for element-wise multiplication with summation between two matrices, then

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 2 \\ 0 & 2 & 2 \end{bmatrix} \star \begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} = 0 \cdot (-1) + 0 \cdot 1 + 0 \cdot (-1) + 0 \cdot (-1) + 2 \cdot 1 + 2 \cdot -1 + 0 \cdot (-1) + 2 \cdot 0 + 2 \cdot 1 = 2$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix} \star \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & -1 \end{bmatrix} = 3$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \star \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix} = 0,$$

and so $2+3+0+1 = 6$, which is the top left element of the output. The next calculation would then involve striding the filter two places to the right over the input and getting -3:

$$\begin{bmatrix} 0 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & 0 & 1 \end{bmatrix} \star \begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} = -3$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 2 \end{bmatrix} \star \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & -1 \end{bmatrix} = -1$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix} \star \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix} = 0,$$

and so $-3 + (-1) + 0 + 1 = -3$. Once the eight strides are completed for the first filter, then you would move on to the second filter and perform the same process to get the second output matrix.

The pooling operation is generally much simpler than the convolution operation. Pooling more or less exists to reduce the size of the features to reduce the amount of compute and prevent overfitting. CNNs typically employ what is known as "max pooling", which down-samples the input by taking the max value for each iteration of the sliding filter. For example, if we have a filter of size 2 and stride 2 and apply it to the following slice of the input (one channel) we get

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}.$$

In this case, you are essentially splitting the matrix into four 2×2 matrices and taking the max element from

each sub-matrix.

3.1.1 VGG Neural Networks

The VGG neural network is a CNN that was proposed in 2014 by K. Simonyan and A. Zisserman from the University of Oxford in the paper *Very Deep Convolutional Networks for Large-Scale Image Recognition* [53]. It was created with the intention to improve upon the AlexNet neural network [8]. The VGG network was designed to be deeper than AlexNet, which at the time was considered the state-of-the-art for ConvNets. Additionally, it was made using smaller convolutional filters in order to increase the receptive field of each neuron, and also to use more pooling layers in order to reduce the dimensionality of the input data. The network is composed of 16-19 layers of convolutional and fully-connected layers. It has been very successful in a number of image classification tasks and has been used to achieve state-of-the-art results in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [53].

3.1.2 Residual Neural Networks

The ResNet architecture was originally proposed in 2015 by Kaiming He, et al. in the paper *Deep Residual Learning for Image Recognition* [3]. The architecture was designed to address the problem of vanishing gradients in very deep neural networks. This can make it difficult or impossible for the algorithm to learn from the data. The problem occurs when the error gradients in the backpropagation algorithm become smaller and smaller as the algorithm moves from the input layer to the hidden layers. This is because the derivatives of the activation function used in the hidden layers tend to be small. The problem is compounded by the fact that the weights in the hidden layers are usually initialized to small random values. ResNet addresses the vanishing gradient problem by using what is known as a skip connection. A skip connection is a connection between two layers that bypasses one or more intervening layers. This allows the gradient to flow directly from the input layer to the output layer, without having to go through the hidden layers. This can significantly improve the training of deep neural networks. It is a small change to the CNN architecture yet with considerable impact. An ensemble of these ResNets was able to win 1st place on the ILSVRC 2015 classification task [3].

3.2 Transformers

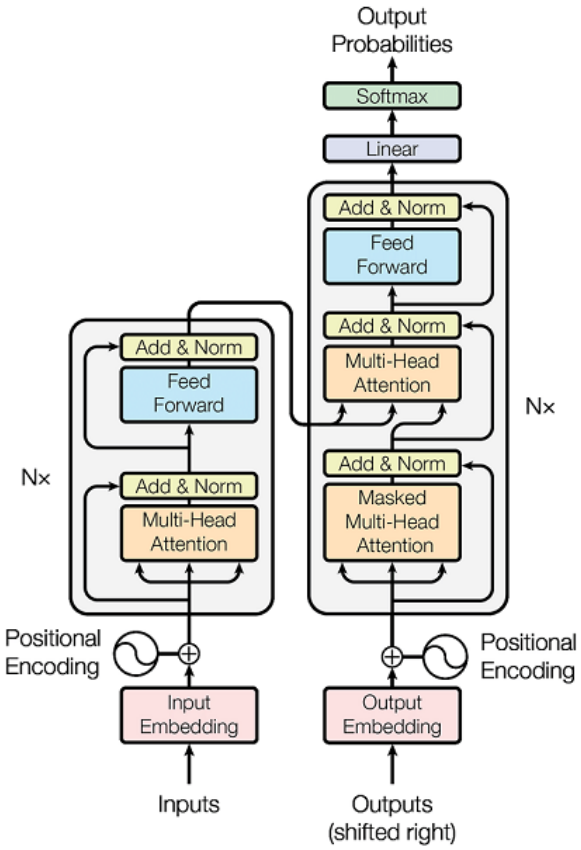


Figure 3.2: Transformer architecture [5]

The Transformer as seen in Figure 3.2 was introduced in 2017 as a response to the failings of RNNs in the space of long-range attention. Attention, in simple terms, is a method by which a neural network manages the interdependence between elements in a sequence. It's essentially how the network tracks the context surrounding an element in the sequence. Additionally, attention also tracks the interdependence between elements in two different sequences (i.e. the input and output sequences). To give an example, take the sentence "How was your day" which you would like to translate into French - "Comment se passe ta journée". The attention mechanism will generate features for the input sentence (English) for each word and these features will provide information for each word itself as well as the words around it. The attention mechanism will then generate a set of weights pertaining to each word in the input sentence for each word in the output sentence. These weights are essentially a representation of how much "attention" the translation should pay attention to each word in the input sentence when generating a word in the output sentence. Each word generated in the translation is not only dependent on the words in the input sentence, but also the words preceding it in the output sentence. The Transformer employs a modified form of attention known as scaled dot-product attention which operates using a system of keys, values, and queries.

Attention forms the basis of the Transformer, hence the title of the paper which introduced it to the

world: *Attention Is All You Need* [5]. However, attention is not exactly "all you need" as the paper may imply. There are several other moving parts in the architecture, namely the encoding-decoding mechanism, the feed-forward networks, and the normalization layers.

3.2.1 Vision Transformer

The original Transformer architecture was developed with the idea of natural language processing in mind. As it so happens, the architecture designed for natural language processing did not handle vision problems very well. In response, the Vision Transformer (ViT) as seen in Figure 3.3, was developed with the goal of introducing the Transformer architecture to the world of computer vision [4].

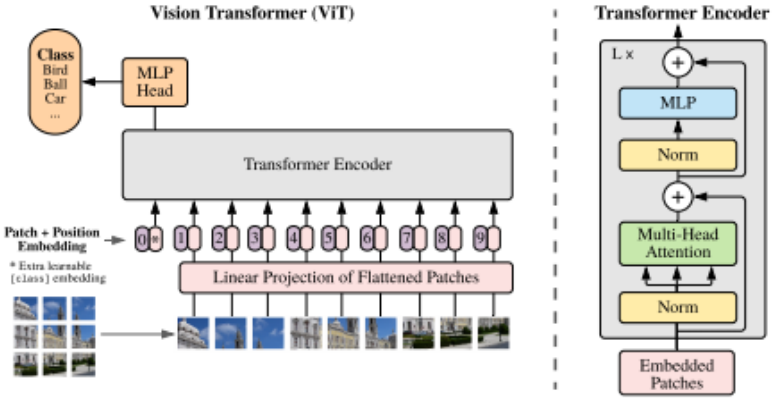


Figure 3.3: Vision Transformer architecture [4].

The Vision Transformer, while not identical to the original Transformer, shares many of the same fundamental building blocks; namely LayerNorm, multi-headed attention, feed-forward artificial neural networks (referred to as multilayer perceptrons or MLP in the Vision Transformer paper) [4].

In the paper, it was found that the Vision Transformer did not quite reach the same level of performance in classification as ResNets of comparable size when training on mid-sized datasets such as ImageNet [41] without strong regularization [4]. However, when the models were trained on larger datasets (14M-300M images), the Visual Transformer approached or even outperformed the ResNets. This was attributed to Transformers lacking some of the inductive biases seen in convolutional neural networks such as translation equivariance and locality. This means that Transformers tend not to generalize as well as CNNs when trained on smaller amounts of data.

The Vision Transformer can essentially be decomposed into three different parts: the image handling at the beginning of the model, the encoder, and the fully-connected output layer. The encoder consists of a stack of encoder blocks. Each block contains two normalization layers (the default is LayerNorm), a self-attention layer, a dropout layer, and an MLP block. The encoder block also possesses skip connections, similar to the ResNet. The normalization layer normalizes the activations of the previous layer. The self-attention

layer allows the model to attend to different parts of the input sequence simultaneously. The dropout layer randomly drops out neurons during training in order to prevent overfitting. The MLP block contains two linear layers, a GELU (Gaussian Error Linear Unit) activation layer, and two dropout layers.

The Vision Transformer works by using a convolutional layer to project the image such that $(c, h, w) \rightarrow (t, n_h, n_w)$, where t is the size of the output of the convolutional layer, $n_h = h//p$ and $n_w = w//p$ where p is patch size and $//$ is the floor division operation. The output from the convolutional layer is then reshaped such that $(t, n_h, n_w) \rightarrow (t, (n_h \times n_w))$. Once the input has been projected and reshaped, it is then sent through the encoder which handles the features of the image and sends its output to the fully-connected output layer which then classifies the input.

The key operation within the encoder is the attention mechanism. The attention mechanism works by calculating the relevance of a set of features from the input with a set of other features from the input. This is done by computing a dot product between a query tensor Q and a key tensor K. These two tensors represent features in the input image. The relevance scores are calculated by $\mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$ where d_k is the dimension of K and acts as a scaling factor to prevent exploding values. The scores are then multiplied by a feature tensor V (known as the value tensor). The product of the score and the value tensor produce a tensor that informs the model which positions in the input image are "important" features, since the features in V are scaled by the values from the softmax. Hence the full attention operation is

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

The multi-head attention block consists of a set of attention mechanisms that look at different subsets of features in the input. The outputs from each attention head are concatenated so that the multi-headed attention block outputs a set of attended features to pass through the normalization layer, which has its output passed through the fully-connected layer and then finally through the dropout layer at the end of the encoder.

Chapter 4

Experiments

4.1 Methods

In this section, we detail the setup for our experiments. The goal behind our experiments is to investigate if there exist any identifiable trends for the different normalization methods across an array of different training schemes with respect to the various metrics employed. The different model configurations can be found in Appendix B. Our experiments are implemented using PyTorch and code is available at https://github.com/gxh2932/Thesis. The data obtained from our experiments can also be found on the GitHub.

To collect our results, we use a ResNet, a VGG-like convnet, and a Vision Transformer, each paired with every normalizer. All the models are trained from scratch using randomly initialized weights per PyTorch's default randomization scheme. Our goal behind employing the different normalizers with different types of models is to filter for results that may be the product of similar architecture. Common results across different models for the same normalizer are more likely to be a result of the normalizer itself. The models are trained using a series of several different hyperparameter configurations. The reasoning behind this follows the same logic as for employing the normalizers across different models. Common results across different hyperparameter configurations are more likely to be a product of properties associated with the architecture and normalizer. Once trained, the class cosine similarity metric is generated for each model by computing the average of the cosine similarities between the average of each class's features. Finally, the Hessian for the model is computed and then used to calculate its trace and its eigenvalue spectral density. For each model, we used two A100 GPUs as provided by RIT Research Computing [54] to collect our results.

Architectures To conduct our experiments we employ a ResNet-56 model, a VGG-10 model, and a Vision Transformer with 6 encoder blocks, each with 12 attention heads.

Model	Param. Count
R56	3.5M
VGG	940K
ViT	11.4M

Data We use both CIFAR-10 and CIFAR-100 [42] to train our models. Each model configuration has two models: one trained on CIFAR-10 and one trained on CIFAR-100. No data augmentations are applied (e.g. image rotation or mirroring). The samples are transformed to have a mean/std of 0.5 in all dimensions, following standard practice with CIFAR datasets.

GroupNorm For experiments involving GroupNorm, we use 11 different GroupNorm configurations which are differentiated by group size. We use group sizes of 2, 4, 8, 16, 32, 64, layer-width$/1$, layer-width$/8$, layer-width$/16$, layer-width$/32$, layer-width$/64$, and layer-width/10000000. Layer width refers to the output dimension of the previous layer which feeds into the GroupNorm layer. Since the layer width can be subject to change throughout the model, GroupNorms that are dependent on layer width are dynamic and do not have a constant group size. It should be noted that for the ResNet, a group size of 64 is not compatible with the standard architecture due to how the ResNet manipulates the dimensions of the features during the throughput process.

Learning Rate We use a learning rate scheduler similar to that of [6, 55], in which the learning rate is scaled linearly with the batch size. The reasoning behind this is that it has been observed that when linearly scaling the learning rate by batch size, the train and test accuracy between using small and large batches remains relatively unchanged [55]. The first set of epochs uses a learning rate of $0.1/(256/\text{bsize})$ and the second uses a learning rate of $0.01/(256/\text{bsize})$. For a batch size of 16, the first set is comprised of 8 epochs and the second is comprised of 2 (10 epochs total). For a batch size of 256, the first set is 40 epochs and the second is 20. We only use batch sizes of 16 and 256.

Momentum & Weight Decay We categorize the experiments into two different sets: experiments that include momentum and weight decay, and experiments that do not. A weight decay of 0.0001 and momentum 0.9 is used for experiments that include them. Outside of the presence of weight decay and momentum, both sets of experiments will be run with the same hyperparameter sweeps. The reason for the distinction is that we wish to investigate the inclusion of weight decay and momentum on the relationship between metrics of flatness and generalization, since it has been found that employing non-vanilla SGD optimizers can weaken the strength of the correlation between flatness and generalization [49]. In other words, we wish to see if there exist differences in the metrics for models trained using vanilla SGD versus those that were not.

4.1.1 Metrics

In this section, we outline the metrics used to gauge different properties of the trained models.

Class Cosine Similarity

We are interested in how each normalizer may influence the cosine similarity between class features. The reason we are choosing to look at class cosine similarity is that we suspect there is potentially some correlation between how well a model generalizes and how well it generates distinct features between classes. We suspect that if a model is proficient at generating distinct features for different classes then perhaps the model will have a less difficult time discerning classes during prediction than otherwise. This investigation is inspired by the result from Lubana et al. [6] in which they showed that models with normalizers which induce dissimilar activations tend to converge faster.

We also suspect that models which generate dissimilar features could be associated with flat minima, since perturbations in the weight space may not have a large impact on the model's ability to generate distinct features (e.g. if you perturb a weight it won't have a big impact on the distinctness of the generated features since there are other distinct features unaffected by the perturbation) and if the perturbations don't result in a significant change in loss after we've reached a minimum it should imply that our minimum is relatively flat.

To compute the cosine similarity between class features, we calculate the average of the features for each class and then take those averaged features to compute the cosine similarity between each class. To get the features, we extract them right before the final fully-connected layer used for classification. Once the cosine similarities between each class are computed, we then average the cosine similarities to find the average cosine similarity of the average features between each class. We use the training set to calculate this metric.

Eigenvalue Mean & Variance We wish to inspect the mean and variance of the eigenvalues calculated using the method outlined in Section 2.3.2. This will give us information with regard to the distribution of eigenvalues for the Hessian matrix.

Greatest Eigenvalue There have been studies that suggest that the greatest eigenvalue of the Hessian may be a good predictor for how well a model generalizes [35, 56, 39, 57]. In essence, the maximum Hessian eigenvalue measures the worst-case loss increase under a perturbation to the weights. However, there have also been counter-studies which show that the greatest eigenvalue may in fact be a weak indicator, if one at all [50, 44]. We wish to contribute to this ongoing debate by inspecting how the magnitude of the greatest eigenvalue correlates with model performance.

Generalization Ratio We are interested in how well models generalize relative to how well they perform in training. We measure this by inspecting the ratio between test and train accuracy (i.e. GR $=$ test/train).

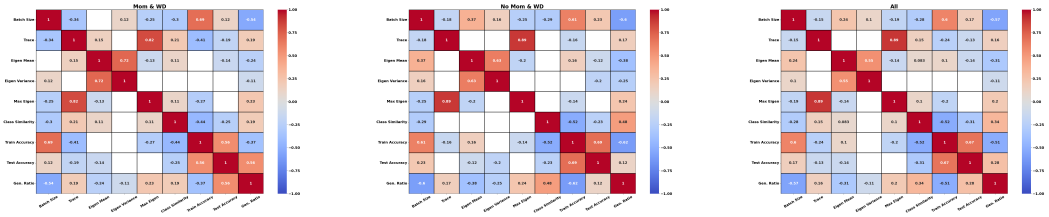
Trace As discussed in Section 2.3.1, we wish to inspect the absolute trace of the Hessian matrix to inspect the "flatness" of the loss surface.

Correlation between metrics/hyperparameters is measured using the Pearson correlation coefficient. The p-values for the respective correlations are computed using the probability density function of the sample correlation coefficient

$$f(r) = \frac{\left(1 - r^2\right)^{n/2-2}}{\mathrm{B}\left(\frac{1}{2}, \frac{n}{2} - 1\right)},$$

where n is the number of samples and B is the beta function. The p-value is a two-side p-value, and is the probability that abs(r') of a random sample x' and y' drawn from the population with zero correlation would be greater than or equal to abs(r). The p-value is calculated by computing $p = 2f(-\text{abs}(r))$. Correlations with p-values less than 0.05 are kept in the heatmaps.

4.2 Results



(a) Mom & WD (b) No Mom & WD (c) All

Figure 4.1: Heatmaps of correlations between different hyperparameters and metrics for models which employ weight decay and momentum, models which do not, and both. Axis labels from left-right/top-down: batch size, trace, eigenvalue mean, eigenvalue variance, max eigenvalue, class similarity, train accuracy, test accuracy, generalization ratio.

Interestingly, class similarity was seemingly the strongest predictor for train accuracy, test accuracy, and generalization ratio across all metrics (trace, eigen mean, eigen variance, maximum eigenvalue, class similarity) as can be seen in Figure A.6.

Overall, BatchNorm and EvoNormBO were the champions of the normalizers. On the other end, InstanceNorm and Plain (identity layer which does not normalize the data) were among the worst across the experiments fairly consistently. Plain configurations suffered from gradient explosion which would render the model untrainable after a few epochs. Even if the model did not suffer gradient explosion, it typically

would still have trouble to converging to a good minimum. InstanceNorm configurations, while more stable, also had trouble converging to good minima. Additionally, the only models which failed to train were those without normalizers. Interestingly, BatchNorm and EBO had the smallest class similarity by a fair margin. They also had the second and third largest traces behind VarNorm, as well as the second and third largest maximum eigenvalues behind VarNorm. Notably, BatchNorm had one of the highest averages for max eigen.

Excluding Plain, the normalizer with the worst test accuracy which used momentum and weight decay (InstanceNorm) was still better than the best which did not (EvoNormBO). This was also true for batch size. The worst performing batch size with momentum and weight decay (16) was better than the best performing batch size without (256). This was not true for the different architectures though, since ResNet and VGG without momentum and weight decay performed better than ViT with.

One of the more interesting takeaways was the performance of Vision Transformers using LayerNorm - the standard normalizer for the architecture. Overall, it performed below average and even went so far as being the *worst* normalizer (outside of Plain) for experiments using momentum and weight decay. This result likely warrants further investigation into the performance of ViT when paired with LayerNorm. Another interesting observation with respect to ViT was the dissimilarity of the features generated by ViT models using BatchNorm. Out of all the different combinations of models trained, ViT + BatchNorm generated the most dissimilar features *by far*. It was also the best performing ViT model overall when considering test accuracy, train accuracy, and stability. This goes against the trend of class similarity being positively correlated with test accuracy for ViT.

When comparing models that were trained to convergence (95% training accuracy or greater) with those that were not, there are several notable differences that emerge. For example, the models that were trained to convergence typically used a batch size of 256, while the non-converged models used a batch size of 64. This difference in batch size may have contributed to the observed differences in the models' behavior. Also, none of the models which converged used CIFAR-100. Only models trained on CIFAR-10 converged. This is not that surprising considering that the CIFAR-100 dataset is more complex due to having more classes [42].

One interesting difference between the converged and non-converged models was the magnitude of the absolute trace. The converged models had a much smaller Hessian trace, approximately 3.8 times smaller, than the non-converged models. This suggests that the converged models settled into flatter minima, which may have contributed to their better generalization performance. Another notable difference between the converged and non-converged models was the cosine similarity between the weights of the models. The converged models had a much smaller cosine similarity, approximately 0.5 versus 0.7 for the non-converged models. Despite these differences, the test accuracy of the converged (61%) and non-converged models

(54%) was somewhat similar.

Architectures ResNet was by far the least flat, with a trace larger than ViT and VGG by an order of magnitude. It also had significantly greater class similarity. Overall, it was the best performing model from a test accuracy perspective, even though it had the lowest train accuracy. This finding is evidence against the hypothesis that flat minima tend to generalize better.

Batch Size Models trained with a batch size of 16 had greater trace, and much greater max eigenvalue, class similarity, and generalization ratio than those trained with a batch size of 256. They also had significantly smaller eigen mean, eigen variance, train accuracy, and somewhat smaller test accuracy.

Group Size For CNNs, group size did not have a large impact on train and test accuracy (less than 3% spread for both ResNet and VGG) outside of group size $= 1$, in general. However, for the Vision Transformer, group size seemingly had a somewhat greater impact on train accuracy ($\sim 5\%$ spread). Interestingly, while ViT models employing group size $= 1$ still performed the worst with respect to test accuracy, there was not a large discrepancy in performance between them and the models using other group sizes like for CNNs. Additionally, they actually had greater train accuracy than some of the other grouping configurations (group size $= 16, 0.25, 0.0000001$).

Skip Connections The inclusion of skip connections in the ResNet led to a much smaller trace (by an order of magnitude), eigen variance, and max eigenvalue as well as smaller eigen mean. It also coincided with improved stability, considering the only ResNet models which failed to train were those which did not use skip connections. Interestingly, the inclusion of skip connections also coincided with lower train and test accuracy. However, when considering models at or near convergence ($> 98\%$ train accuracy), models with skip connections had smaller trace, eigen mean, eigen variance, and max eigen while also having greater train and test accuracy. The class similarity was still greater for these models, however.

Optimization Models trained with vanilla SGD had a much greater trace, max eigenvalue and class similarity. They also had greater eigen mean and significantly smaller eigen variance, train accuracy, and test accuracy.

4.2.1 Without Momentum & Weight Decay

Overall, EvoNormBO, VarianceNorm, and BatchNorm were the top performers with respect to test accuracy. InstanceNorm and Plain were by far the worst performers when you consider the instability of Plain. For both the Vision Transformer and the ResNet, half the models without a normalizer (Plain) failed during

training. When not using a normalizer, the Vision Transformer only failed to train when using a batch size of 256, and the ResNet only failed to train when not using skip connections.

For the Vision Transformer, BatchNorm was the stand-out performer, while Plain and EvoNormSO were generally the worst. Interestingly, the test and train accuracy spread across the different norms was relatively small for ViT compared to the other models. Perhaps the Vision Transformer relies on normalization less than the other architectures. When comparing the successful ViT + Plain models to other ViT models which use a batch size of 16, we see they underperform relative to the competition. For the VGG architecture, VarianceNorm and EvoNormBO were the top performers, while InstanceNorm was by far the worst. For the ResNet architecture, EvoNormBO, BatchNorm, and VarianceNorm were by far the top performers, while InstanceNorm and Plain were the worst. When comparing ResNet-56 + Plain with skip connections (i.e. successful ResNet-56 + Plain models) to other ResNet-56 models employing skip connections, Plain was relatively average with regard to test accuracy.

Among models which used vanilla SGD, VGG had the highest test accuracy, ResNet-56 second, and ViT third. For training accuracy, ViT $>$ VGG $>$ ResNet-56 (i.e. ViT converged to a solution the quickest). In terms of trace, ResNet-56 $>$ ViT $>$ VGG (i.e. VGG was the flattest). In terms of class similarity, ResNet-56 $>$ VGG $>$ ViT (i.e. ViT produced the least similar features between classes). For the generalization ratio, ResNet-56 $>$ VGG $>$ ViT (same ordering as the class similarity metric).

4.2.2 With Momentum & Weight Decay

BatchNorm and EvoNormBO were the clear top performers with respect to test accuracy for training configurations that used momentum and weight decay, while InstanceNorm and Plain were the clear worst performers. This follows the trend seen in configurations with vanilla stochastic gradient descent. In fact, BatchNorm and EvoNormBO were the clear top performers for all three architectures. Once again, for both the Vision Transformer and the ResNet, half the models without a normalizer failed during training. When using no normalizer, the Vision Transformer only failed to train when using a batch size of 256, and the ResNet only failed to train when not using skip connections. Interestingly, Vision Transformers without a normalizer had the highest test accuracy amongst training configurations with a batch size of 16.

For the Vision Transformer, GroupNorm (w/ layer-width 32 and 64), BatchNorm, EvoNormBO were the stand-out performers. Once again, the test and train accuracy spread across the different norms was relatively small compared to the other models. When comparing the successful Plain models to other models which use a batch size of 16, we see that it actually performs *significantly* better than the competition. For the VGG architecture, EvoNormBO, BatchNorm, and EvoNormSO were the top performers, while InstanceNorm was by far the worst. For the ResNet architecture, BatchNorm was the stand-out performer, while EvoNormSO and Plain were the worst. When comparing ResNet-56 + Plain with skip connections to other ResNet-56

models employing skip connections, Plain was once again relatively average with regard to test accuracy.

Among models which did not use vanilla SGD, ResNet-56 had the highest test accuracy, VGG second, and ViT third. For training accuracy, VGG $>$ ResNet-56 $>$ ViT. In terms of trace, ResNet-56 $>$ VGG $>$ ViT. In terms of class similarity, ResNet-56 $>$ ViT $>$ VGG. For the generalization ratio, ResNet-56 $>$ VGG $>$ ViT.

Chapter 5

Discussion & Future Work

On a slightly less rigorous note, I think it's probably the case that metrics of flatness only provide a narrow window into the connection between the geometry of the loss surface and how well the model generalizes. It ignores the question of depth, which is just as important as flatness. A flat but shallow surface won't generalize nearly as well as a deeper surface with the same level of flatness. That's not to say flatness is a bad metric however when you consider the nature of how deep nets optimize, since stochastic gradient descent tends to favor deeper basins over shallower basins, especially in high dimensions. Good minima seem to generally be some type of high-volume basin with a flat bottom. Deepness implies low loss while flatness implies robustness to parameter perturbations, and hence larger decision boundaries.

Randomness, particularly with respect to the model parameters, also plays a very important role in how stochastic gradient descent traverses the loss surface and locates minima. It's standard practice to initialize your deep nets with random weights at the beginning of training. From the perspective of the loss surface, this is essentially picking a random point on the surface from where to start your traversal. Your starting point is going to influence how your model converges. Different initializations have a chance to lead you toward different minima basins. And of course not all basins are created equal, so the level of generalization and flatness your model will achieve will be fairly random even when training on the same data with the same hyperparameters. Ultimately, even if two models have the same exact structure, you're starting with two different functions at the initialization of the optimization process. The presence of this randomness makes it difficult to draw strong conclusions from studies such as the one done in this paper, since it pushes toward the need for large sample sizes, and even relatively small models with only a few million parameters require exceptionally large amounts of compute. This is especially true when estimating something second-order like the Hessian.

On the topic of class cosine similarity; intuitively I imagine there might be some optimal range of cosine similarity of features, dependent on the model/hyperparameters. My reasoning for this is that I imagine

if a model generates features that are too dissimilar/similar that there is probably some type of overfitting/underfitting occurring. Empirically it was the case that the class similarity was almost always somewhere between 0 and 1, and the average class cosine similarity for ResNet/VGG/ViT was $0.713/0.579/0.572$ respectively. When you consider that cosine similarity exists in a range from -1 to 1, these are fairly small gaps. Especially between VGG and ViT which almost have an identical average.

Building off of this idea of a potential relationship between the class cosine similarity and goodness of fit, perhaps comparing training schemes where the number of parameters is less than the number of training samples with schemes where the number of parameters is greater. Or in other terms, comparing the results of training schemes on different sides of the interpolation threshold. Perhaps the class cosine similarity and the metrics of flatness would behave differently on opposing sides of the interpolation threshold. It would also be interesting to inspect if some normalization techniques performed differently on the different ends of the threshold as well.

Examining the actual feature distributions themselves may provide more insight into why some normalization techniques work better than others. When training a model with different normalization techniques, the distributions of features throughout the network are likely to be different due to the way that the normalization techniques transform the data. This can potentially have a significant impact on the network's behavior and performance. Similarly, it is possible that other normalization techniques may also have an impact on internal covariate shift. While the internal covariate shift argument may not be as strong as it was prior to the work done by Santurkar [18], perhaps there is still some merit to investigating internal covariate shift from the perspective of normalization techniques besides just BatchNorm.

One of the decisions made for our training setup was to use a fixed number of epochs rather than just training every model to convergence. This was done mostly for the sake of time since training every model to convergence would take quite a bit longer than just training for a set number of epochs. In future work, it may be worth training every model to convergence and investigating how that may change some of the results.

I would say that one of the biggest limiting factors of this paper was the amount of data I could generate in a reasonable amount of time. The final set of data that I used in this paper included over 700 models, and of course, I trained many more models than this over the course of the debugging process. Given more time, I would like to have fidgeted with the learning rates more to see how that impacted the various metrics. In total, it probably took hundreds to thousands of hours of computing on state-of-the-art GPUs just to acquire 700 samples, which isn't even that large of a dataset. I think this soft bottleneck outlines the importance of theoretical work in the field, since there is only so much computing you can do to collect empirical results. Not to mention the concern of how large amounts of computing can strain finances and potentially have negative impacts on the environment. That's not to say empirical work is bad. In fact, I think it's very

important, since observations can lead to a more rigorous formulation of the problem. It's also the case that the theory behind the problems presented in this thesis–and for deep learning in general–seem to be lagging behind the empirical results. It's why the work from someone like Cybenko with his proof of the *Universal Approximation Theorem* [58] or Belkin with his work on *Double Descent* [59, 60] is so important. Until the theory does catch up, we won't know *why* things work, just how they *may* work, and the former can be so much more powerful than the latter.

Bibliography

[1] S. J. Russell and P. Norvig, *Artificial Intelligence: A modern approach*, 3rd ed. Prentice-Hall, 2010.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014. [Online]. Available: https://arxiv.org/abs/1409.4842

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[4] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2020. [Online]. Available: https://arxiv.org/abs/2010.11929

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," vol. 30, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[6] E. S. Lubana, R. P. Dick, and H. Tanaka, "Beyond batchnorm: Towards a unified understanding of normalization in deep learning," 2021. [Online]. Available: https://arxiv.org/abs/2106.05956

[7] N. Pinto, D. D. Cox, and J. J. DiCarlo, "Why is real-world visual object recognition hard?" *PLOS Computational Biology*, vol. 4, no. 1, pp. 1–6, 01 2008. [Online]. Available: https://doi.org/10.1371/journal.pcbi.0040027

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," vol. 25, 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[10] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016. [Online]. Available: https://arxiv.org/abs/1607.06450

[11] Y. Wu and K. He, "Group normalization," 2018. [Online]. Available: https://arxiv.org/abs/1803.08494

[12] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," 2016. [Online]. Available: https://arxiv.org/abs/1607.08022

[13] S. Singh and S. Krishnan, "Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1911.09737

[14] H. Daneshmand, J. Kohler, F. Bach, T. Hofmann, and A. Lucchi, "Batch normalization provably avoids rank collapse for randomly initialised deep networks," 2020. [Online]. Available: https://arxiv.org/abs/2003.01652

[15] H. Liu, A. Brock, K. Simonyan, and Q. V. Le, "Evolving normalization-activation layers," 2020. [Online]. Available: https://arxiv.org/abs/2004.02967

[16] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_3

[17] I. Rajagopal, "Batch normalization - speed up neural network training," https://medium.com/@ilango100/batch-normalization-speed-up-neural-network-training-245e39a62f85, Jun. 2018.

[18] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" 2018. [Online]. Available: https://arxiv.org/abs/1805.11604

[19] S. Wiesler and H. Ney, "A convergence analysis of log-linear training," vol. 24, 2011. [Online]. Available: https://proceedings.neurips.cc/paper/2011/file/e836d813fd184325132fca8edcdfb40e-Paper.pdf

[20] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2016. [Online]. Available: https://arxiv.org/abs/1605.07146

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015. [Online]. Available: https://arxiv.org/abs/1512.00567

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," 2016. [Online]. Available: https://arxiv.org/abs/1603.05027

[23] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," 2016. [Online]. Available: https://arxiv.org/abs/1612.00593

[24] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2016. [Online]. Available: https://arxiv.org/abs/1608.06993

[25] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1611.05431

[26] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1602.07868

[27] B. Neyshabur, R. Tomioka, and N. Srebro, "Norm-based capacity control in neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1503.00036

[28] P. Bartlett, D. J. Foster, and M. Telgarsky, "Spectrally-normalized margin bounds for neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1706.08498

[29] B. Neyshabur, S. Bhojanapalli, and N. Srebro, "A PAC-bayesian approach to spectrally-normalized margin bounds for neural networks," 2018. [Online]. Available: https://openreview.net/forum?id=Skz_WfbCZ

[30] N. Golowich, A. Rakhlin, and O. Shamir, "Size-independent sample complexity of neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1712.06541

[31] V. Nagarajan and J. Z. Kolter, "Deterministic pac-bayesian generalization bounds for deep networks via generalizing noise-resilience," 2019. [Online]. Available: https://arxiv.org/abs/1905.13344

[32] C. Wei and T. Ma, "Data-dependent sample complexity of deep neural networks via lipschitz augmentation," 2019. [Online]. Available: https://arxiv.org/abs/1905.03684

[33] P. M. Long and H. Sedghi, "Generalization bounds for deep convolutional neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1905.12600

[34] S. Hochreiter and J. Schmidhuber, "Flat minima," *Neural computation*, vol. 9, no. 1, pp. 1–42, 1997.

[35] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," 2016. [Online]. Available: https://arxiv.org/abs/1609.04836

[36] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson, "Averaging weights leads to wider optima and better generalization," 2018. [Online]. Available: https://arxiv.org/abs/1803.05407

[37] H. Wang, N. S. Keskar, C. Xiong, and R. Socher, "Identifying generalization properties in neural networks," 2019. [Online]. Available: https://openreview.net/forum?id=BJxOHs0cKm

[38] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," 2017. [Online]. Available: https://arxiv.org/abs/1712.09913

[39] P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, and R. Zecchina, "Entropy-sgd: Biasing gradient descent into wide valleys," 2016. [Online]. Available: https://arxiv.org/abs/1611.01838

[40] G. K. Dziugaite and D. M. Roy, "Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data," 2017. [Online]. Available: https://arxiv.org/abs/1703.11008

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[42] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[43] Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney, "Pyhessian: Neural networks through the lens of the hessian," 2019. [Online]. Available: https://arxiv.org/abs/1912.07145

[44] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, "Sharp minima can generalize for deep nets," 2017. [Online]. Available: https://arxiv.org/abs/1703.04933

[45] W. R. Huang, Z. Emam, M. Goldblum, L. Fowl, J. K. Terry, F. Huang, and T. Goldstein, "Understanding generalization through visualizations," 2019. [Online]. Available: https://arxiv.org/abs/1906.03291

[46] M. Hutchinson, "A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines," *Communication in Statistics- Simulation and Computation*, vol. 18, pp. 1059–1076, 01 1989.

[47] P. Hitczenko and S. Kwapień, "On the rademacher series," 1994.

[48] J. Demmel, "Chapter 7: Iterative methods for eigenvalue problems," in *Applied Numerical Linear Algebra*. SIAM, 1997.

[49] S. Zhang, I. Reid, G. V. Pérez, and A. Louis, "Why flatness does and does not correlate with generalization for deep neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2103.06219

[50] S. Kaur, J. Cohen, and Z. C. Lipton, "On the maximum hessian eigenvalue and generalization," 2022. [Online]. Available: https://arxiv.org/abs/2206.10654

[51] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[52] "What is a convolutional neural network?" https://ww2.mathworks.cn/en/discovery/convolutional-neural-network-matlab.html.

[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition." arXiv, 2014. [Online]. Available: https://arxiv.org/abs/1409.1556

[54] Rochester Institute of Technology, "Research computing services," 2022. [Online]. Available: https://www.rit.edu/researchcomputing/

[55] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 2017. [Online]. Available: https://arxiv.org/abs/1706.02677

[56] Y. Wen, K. Luk, M. Gazeau, G. Zhang, H. Chan, and J. Ba, "An empirical study of large-batch stochastic gradient descent with structured covariance noise," 2019. [Online]. Available: https://arxiv.org/abs/1902.08234

[57] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey, "Three factors influencing minima in sgd," 2017. [Online]. Available: https://arxiv.org/abs/1711.04623

[58] G. V. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989.

[59] M. Belkin, D. Hsu, S. Ma, and S. Mandal, "Reconciling modern machine-learning practice and the classical bias-variance trade-off," *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15 849–15 854, 2019. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1903070116

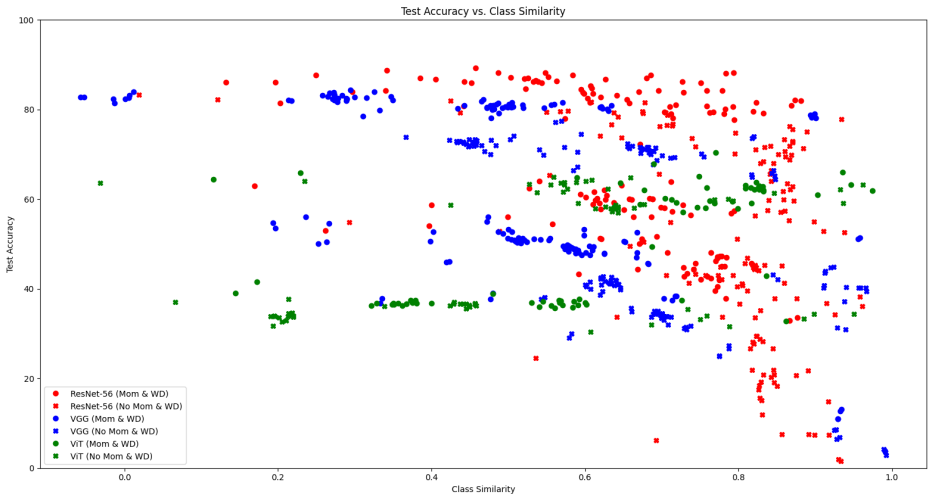
[60] M. Belkin, D. Hsu, and J. Xu, "Two models of double descent for weak features," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 4, pp. 1167–1180, Jan. 2020. [Online]. Available: https://doi.org/10.1137%2F20m1336072

Appendix A

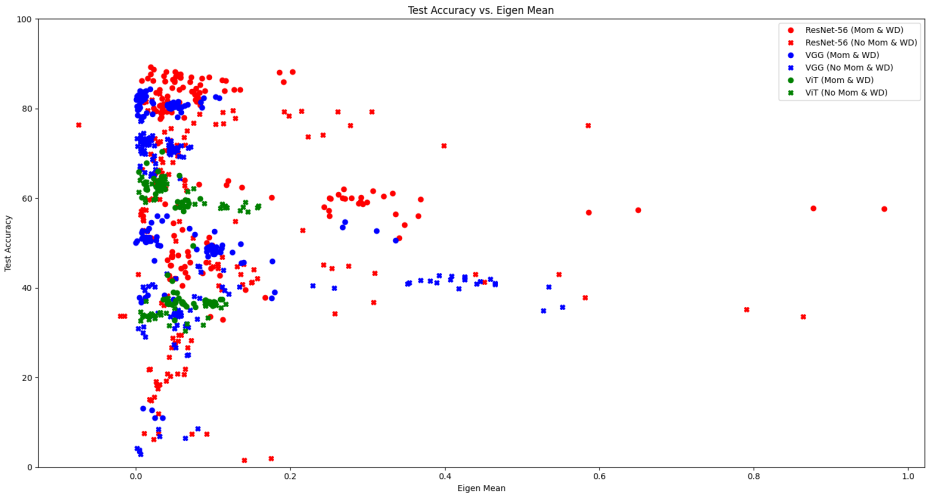
Figures

Below exist the figures showing the results from the conducted experiments. They are broken up into four sections: test accuracy vs. metrics for all the different models and training configurations, results of the experiments for models without momentum & weight decay, results for models with momentum & weight decay, and test accuracy vs. metrics for all the different p-groupings for models which used GroupNorm.

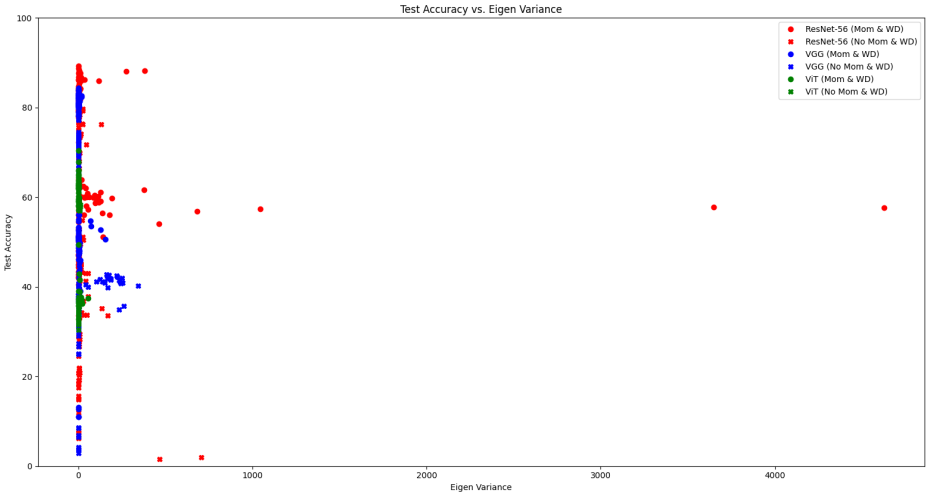
A.1 Test Accuracy vs. Metrics



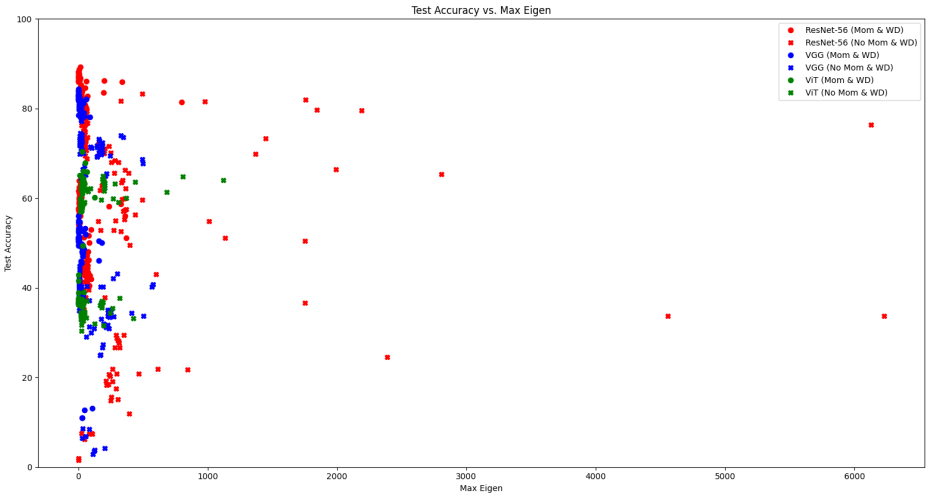
(a) Class Similarity



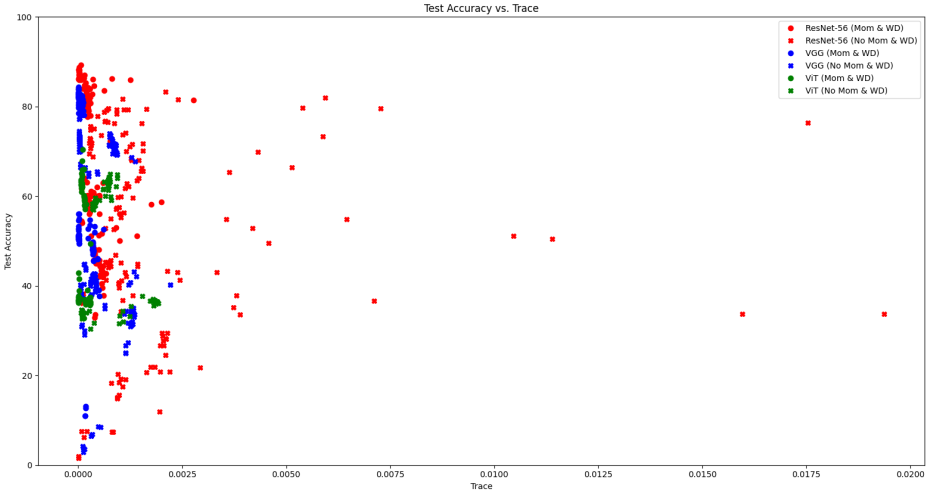
(b) Eigen Mean



(c) Eigen Variance

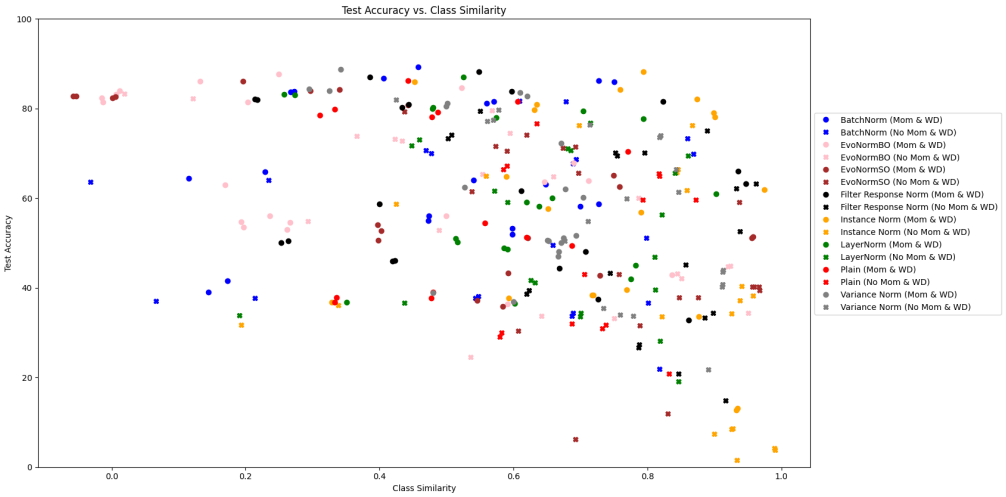


(d) Max Eigen

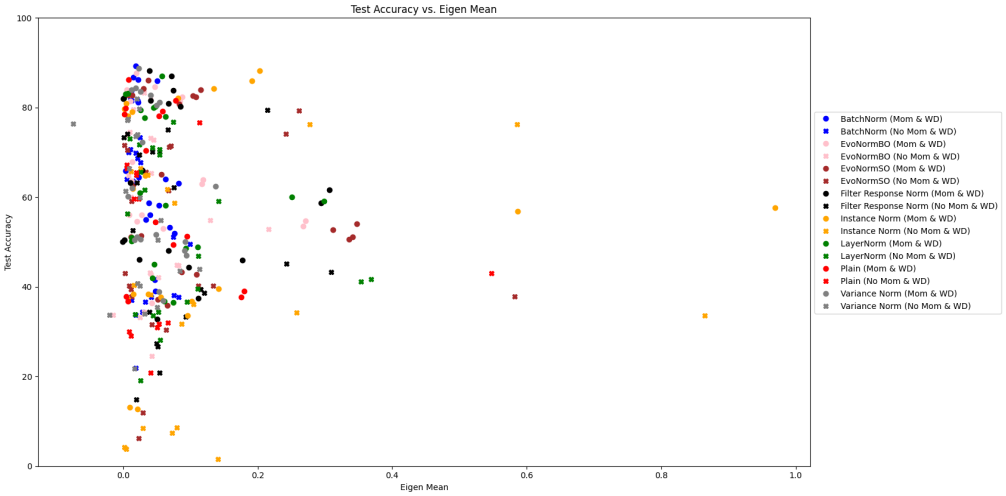


(e) Trace

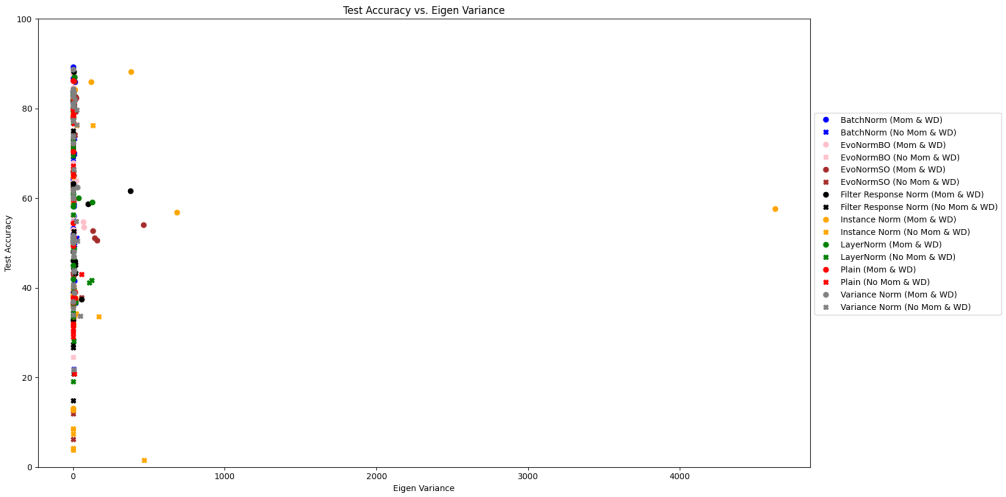
Figure A.1: Test Accuracy of architectures plotted against the investigated metrics.



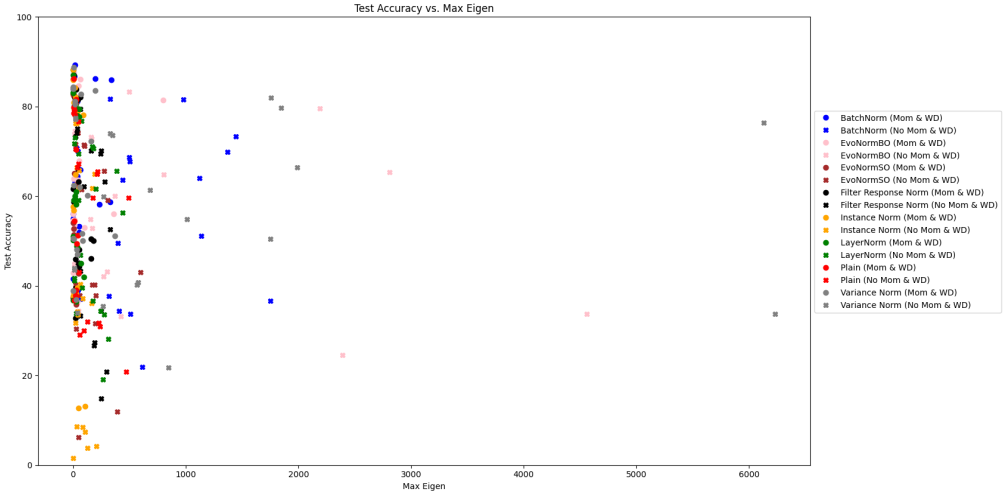
(a) Class Similarity



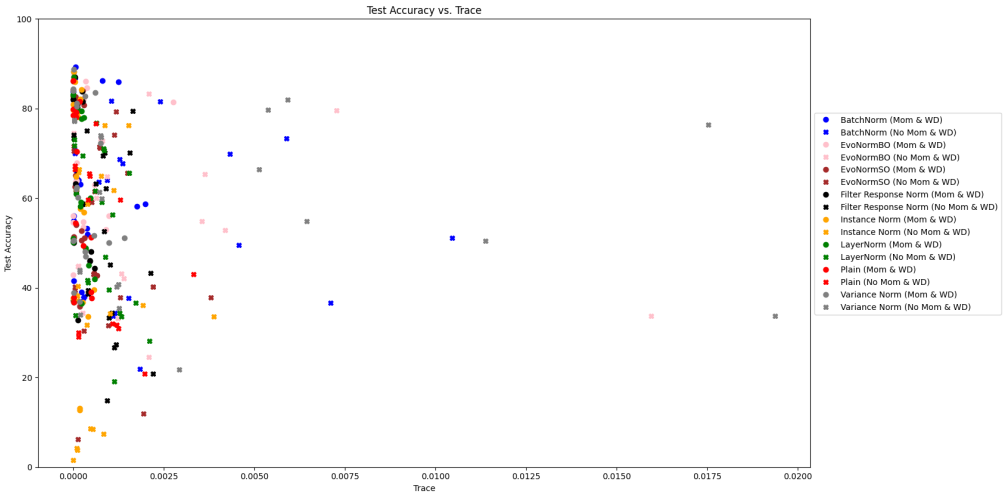
(b) Eigen Mean



(c) Eigen Variance



(d) Max Eigen

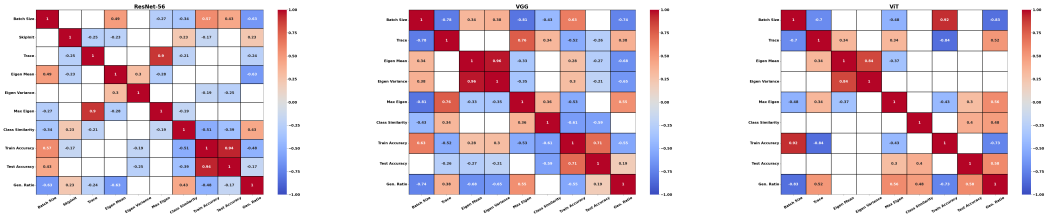


(e) Trace

Figure A.2: Test Accuracy of normalizers (excluding GroupNorm due to overpopulation of graph) plotted against the investigated metrics.

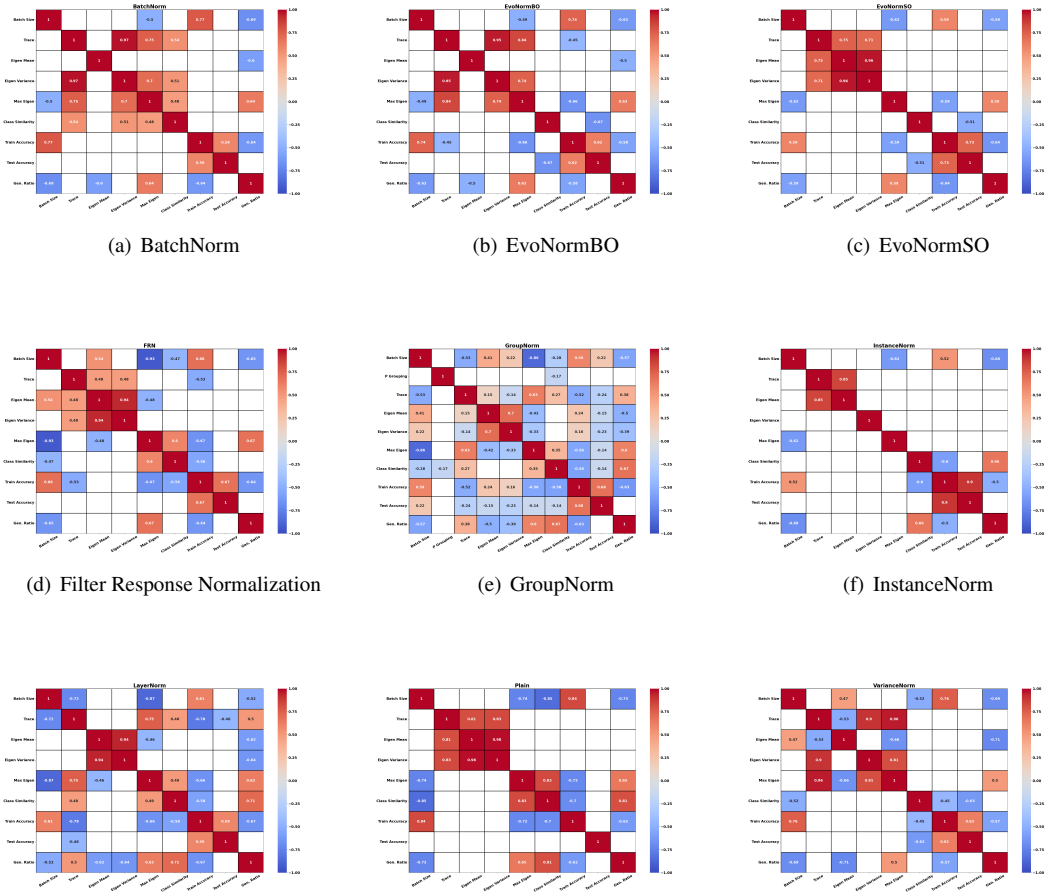
A.2 Without Momentum & Weight Decay

The figures in this section showcase the performance of each normalizer within the respective model. They do not factor in models which failed to train.



(a) ResNet-56 (b) VGG (c) ViT

Figure A.3: Heatmaps of correlations between different hyperparameters and metrics for different model types which did not use momentum and weight decay. Axis labels from left-right/top-down: batch size, skip connections, trace, eigenvalue mean, eigenvalue variance, max eigenvalue, class similarity, train accuracy, test accuracy, generalization ratio.



(a) BatchNorm (b) EvoNormBO (c) EvoNormSO

(d) Filter Response Normalization (e) GroupNorm (f) InstanceNorm

(g) LayerNorm (h) Plain (i) VarianceNorm

Figure A.4: Heatmaps of correlations between different hyperparameters and metrics for different normalizers which did not use momentum and weight decay. Axis labels from left-right/top-down: batch size, trace, eigenvalue mean, eigenvalue variance, max eigenvalue, class similarity, train accuracy, test accuracy, generalization ratio.

(a) Average Across Architectures (no plain)

Norm	Test Accuracy	Final Train Accuracy
EvoNormBO	55.403	78.291
VarianceNorm	54.881	76.245
BatchNorm	53.790	79.932
LayerNorm	50.881	70.411
FRN	50.264	68.658
GroupNorm	49.721	69.917
EvoNormSO	48.560	70.099
InstanceNorm	39.447	57.112

(b) ResNet-56 w/o plain

Norm	Test Accuracy	Final Train Accuracy
EvoNormBO	59.511	74.200
BatchNorm	58.194	75.543
VarianceNorm	58.147	68.636
LayerNorm	50.218	52.884
FRN	50.165	55.854
GroupNorm	49.588	53.664
EvoNormSO	44.461	53.240
InstanceNorm	41.151	49.542

(c) ResNet-56 w/ plain, skipinit=True

Norm	Test Accuracy	Final Train Accuracy
EvoNormBO	56.714	70.409
BatchNorm	56.083	71.927
VarianceNorm	55.658	64.550
Plain	50.025	61.557
LayerNorm	47.931	49.921
FRN	46.886	50.496
GroupNorm	46.507	48.774
EvoNormSO	40.272	47.046
InstanceNorm	37.572	42.637

(d) VGG

Norm	Test Accuracy	Final Train Accuracy
VarianceNorm	58.831	83.556
EvoNormBO	58.624	88.637
EvoNormSO	55.612	87.722
LayerNorm	54.639	80.475
BatchNorm	52.600	84.012
FRN	52.386	77.239
GroupNorm	51.677	76.420
Plain	48.199	77.788
InstanceNorm	29.343	39.280

(e) ViT no plain

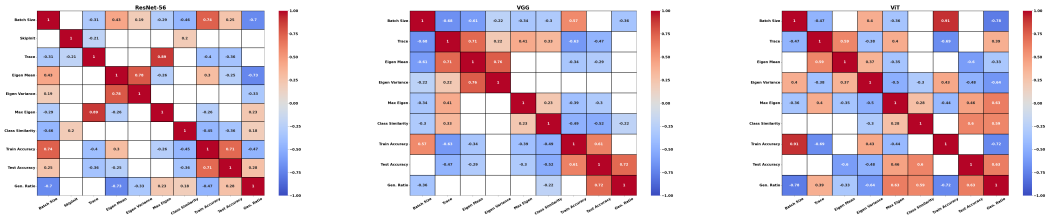
Norm	Test Accuracy	Final Train Accuracy
BatchNorm	50.575	80.240
FRN	48.242	72.880
EvoNormBO	48.075	72.037
GroupNorm	47.898	79.668
InstanceNorm	47.847	82.515
LayerNorm	47.786	77.875
VarianceNorm	47.664	76.543
EvoNormSO	45.606	69.336

(f) ViT w/ plain, batch size=16

Norm	Test Accuracy	Final Train Accuracy
BatchNorm	50.850	61.700
InstanceNorm	50.511	65.277
GroupNorm	49.907	59.492
LayerNorm	49.111	57.182
FRN	48.800	53.041
VarianceNorm	48.422	56.676
EvoNormBO	46.556	51.739
Plain	45.794	52.861
EvoNormSO	45.322	48.516

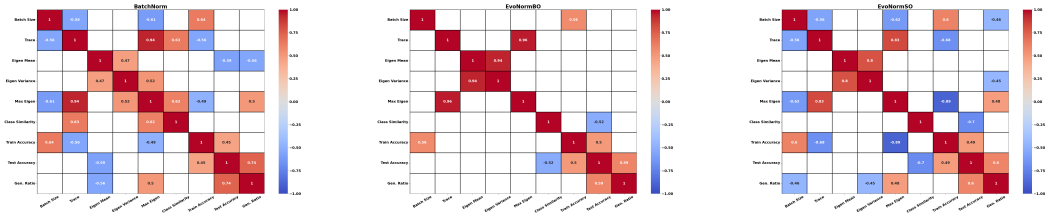
Table A.1: Average performance of every model which did not use momentum and weight decay.

A.3 With Momentum & Weight Decay

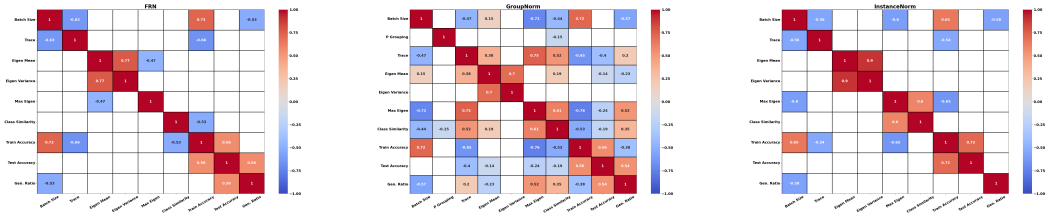


(a) ResNet-56 (b) VGG (c) ViT

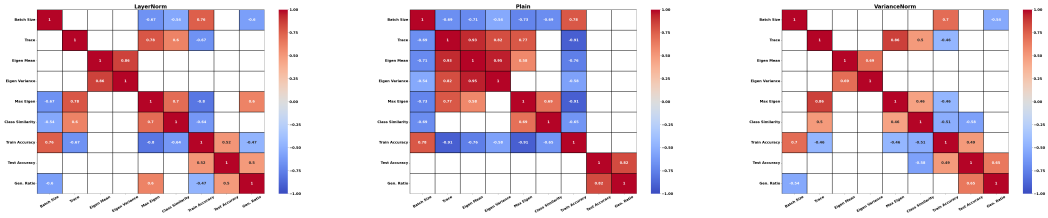
Figure A.5: Heatmaps of correlations between different hyperparameters and metrics for different model types. Axis labels from left-right/top-down: batch size, skip connections, trace, eigenvalue mean, eigenvalue variance, max eigenvalue, class similarity, train accuracy, test accuracy, generalization ratio.



(a) BatchNorm (b) EvoNormBO (c) EvoNormSO



(d) Filter Response Normalization (e) GroupNorm (f) InstanceNorm



(g) LayerNorm (h) Plain (i) VarianceNorm

Figure A.6: Heatmaps of correlations between different hyperparameters and metrics for different normalizers which used momentum and weight decay. Axis labels from left-right/top-down: batch size, trace, eigenvalue mean, eigenvalue variance, max eigenvalue, class similarity, train accuracy, test accuracy, generalization ratio.

(a) Average Across Architectures (no plain)

Norm	Test Accuracy	Final Train Accuracy
BatchNorm	65.015	89.292
EvoNormBO	64.436	86.134
FRN	61.236	83.433
VarianceNorm	61.020	84.118
EvoNormSO	60.994	86.486
GroupNorm	60.391	83.506
LayerNorm	59.900	82.040
InstanceNorm	56.269	77.700

(b) ResNet-56 no plain

Norm	Test Accuracy	Final Train Accuracy
BatchNorm	74.006	90.139
EvoNormBO	71.935	87.358
FRN	69.178	84.552
VarianceNorm	67.806	84.519
GroupNorm	67.513	82.001
LayerNorm	66.015	79.529
InstanceNorm	66.001	82.489
EvoNormSO	65.767	84.752

(c) ResNet-56 w plain, skipinit=True

Norm	Test Accuracy	Final Train Accuracy
BatchNorm	74.228	89.501
EvoNormBO	73.033	89.885
VarianceNorm	71.372	86.744
FRN	69.869	84.827
Plain	68.394	86.500
GroupNorm	67.720	82.726
LayerNorm	67.203	81.721
InstanceNorm	65.164	80.842
EvoNormSO	64.700	84.991

(d) VGG

Norm	Test Accuracy	Final Train Accuracy
EvoNormBO	68.721	97.029
BatchNorm	68.326	96.992
EvoNormSO	67.043	97.899
VarianceNorm	65.756	92.167
LayerNorm	65.622	90.468
FRN	64.706	91.278
GroupNorm	64.399	89.577
Plain	58.362	90.685
InstanceNorm	52.521	76.650

(e) ViT no plain

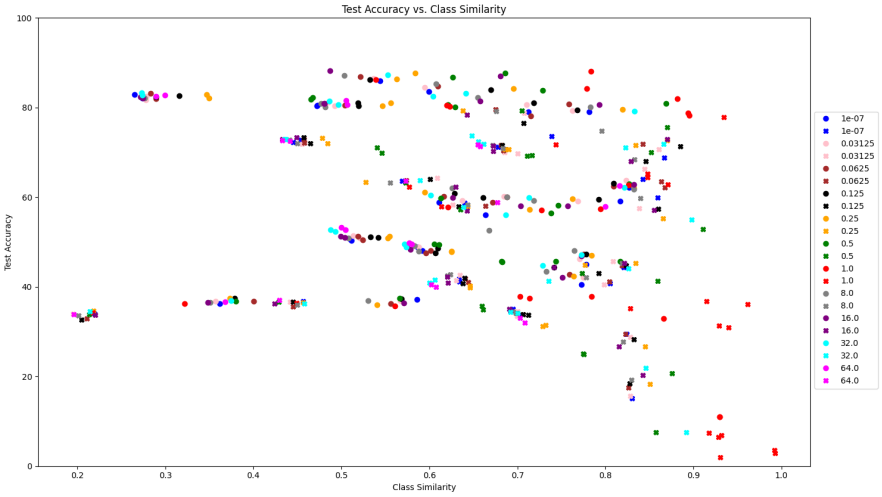
Norm	Test Accuracy	Final Train Accuracy
BatchNorm	52.714	80.747
EvoNormBO	52.653	74.016
InstanceNorm	50.286	73.961
EvoNormSO	50.172	76.808
FRN	49.825	74.469
VarianceNorm	49.500	75.668
GroupNorm	49.262	78.939
LayerNorm	48.064	76.123

(f) ViT w plain, bsize=16

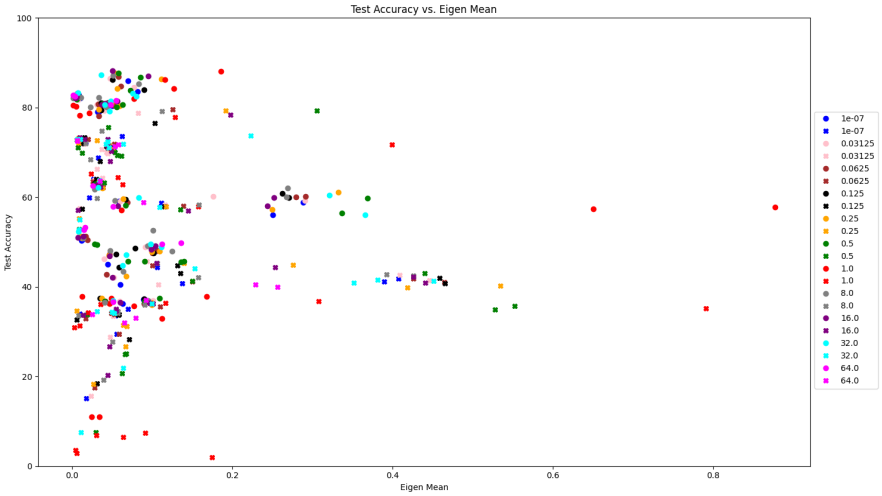
Norm	Test Accuracy	Final Train Accuracy
Plain	59.856	71.727
BatchNorm	52.461	62.840
GroupNorm	50.909	60.063
InstanceNorm	50.783	61.754
EvoNormBO	49.922	55.480
VarianceNorm	49.461	58.377
EvoNormSO	49.244	55.816
LayerNorm	48.661	55.648
FRN	47.983	52.785

Table A.2: Average performance of every model which used momentum and weight decay.

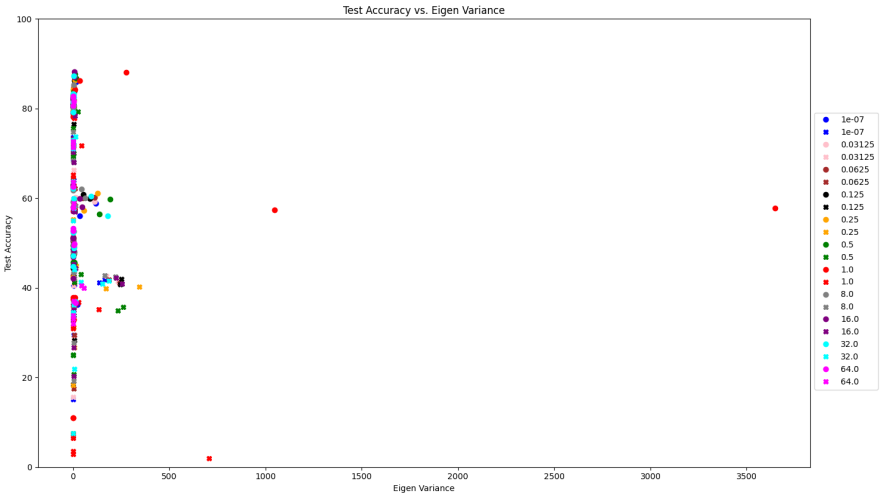
A.4 P-groupings



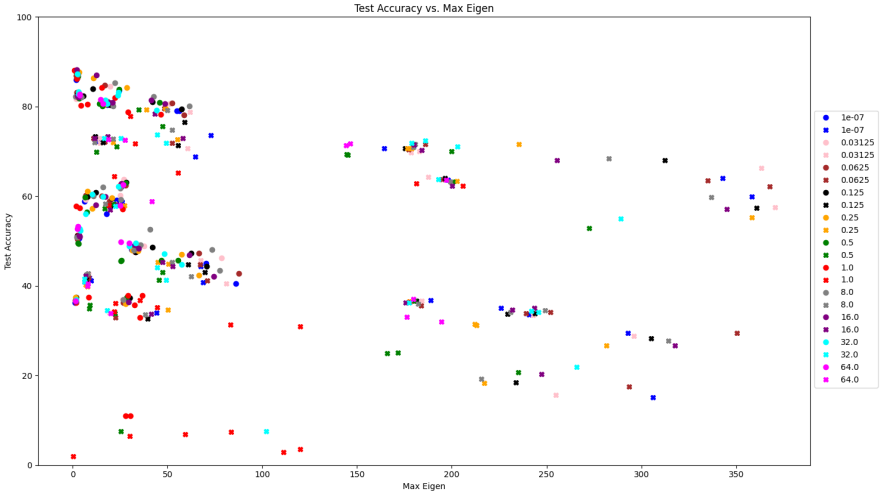
(a) Class Similarity



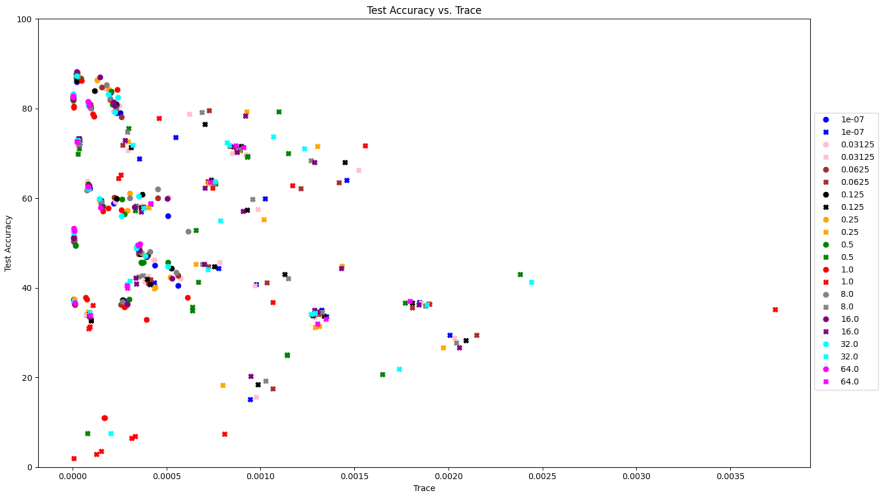
(b) Eigen Mean



(c) Eigen Variance



(d) Max Eigen



(e) Trace

Figure A.7: Test Accuracy of p-groupings plotted against the investigated metrics.

A.4.1 Without Momentum & Weight Decay

(a) Average Across Architectures (no gsize 64)

P-grouping	Test Accuracy	Final Train Accuracy
8.000e+00	51.652	72.083
1.600e+01	51.349	72.214
6.250e-02	51.176	71.449
1.250e-01	51.164	71.444
2.500e-01	51.155	71.665
3.125e-02	51.106	71.095
1.000e-07	50.853	70.848
3.200e+01	50.497	71.904
5.000e-01	48.830	70.503
1.000e+00	38.452	54.280

(b) ResNet-56 P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
8.000e+00	52.051	55.383
2.500e-01	51.737	55.676
1.600e+01	51.622	55.938
6.250e-02	51.246	54.403
1.250e-01	50.971	54.079
3.125e-02	50.481	53.119
1.000e-07	49.504	52.156
5.000e-01	48.765	54.120
3.200e+01	48.290	54.378
1.000e+00	41.217	47.388

(c) VGG P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
8.000e+00	55.133	80.956
3.200e+01	55.125	81.383
1.600e+01	55.114	81.178
1.000e-07	54.811	80.847
6.250e-02	54.721	80.598
1.250e-01	54.718	80.503
3.125e-02	54.557	80.397
6.400e+01	54.204	81.280
2.500e-01	53.632	79.767
5.000e-01	49.996	77.758
1.000e+00	26.438	35.953

(d) ViT P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
6.400e+01	48.308	79.877
3.125e-02	48.281	79.769
1.000e-07	48.244	79.541
2.500e-01	48.094	79.550
3.200e+01	48.075	79.951
1.250e-01	47.803	79.752
8.000e+00	47.772	79.910
5.000e-01	47.728	79.631
1.000e+00	47.703	79.498
6.250e-02	47.561	79.347
1.600e+01	47.311	79.526

Table A.3: Average performance of each p-grouping employed in each architecture for training schemes which did not use momentum and weight decay.

A.4.2 With Momentum & Weight Decay

(a) Average Across Architectures (no gsize 64)

P-grouping	Test Accuracy	Final Train Accuracy
3.200e+01	62.326	87.537
8.000e+00	60.966	83.326
1.250e-01	60.917	83.696
2.500e-01	60.806	83.601
1.600e+01	60.726	83.485
3.125e-02	60.722	83.133
6.250e-02	60.527	83.611
5.000e-01	60.500	84.035
1.000e-07	60.038	82.577
1.000e+00	55.191	78.452

(b) ResNet-56 P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
8.000e+00	68.524	81.836
5.000e-01	68.321	84.042
2.500e-01	68.171	83.029
1.600e+01	68.010	81.929
1.250e-01	67.867	82.476
6.250e-02	67.589	81.430
3.200e+01	67.568	83.134
3.125e-02	67.310	80.329
1.000e-07	65.983	79.918
1.000e+00	65.790	81.889

(c) VGG P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
6.400e+01	66.560	92.118
3.200e+01	66.417	91.630
3.125e-02	65.744	90.912
8.000e+00	65.744	90.841
1.600e+01	65.732	91.296
1.250e-01	65.565	90.825
2.500e-01	65.501	90.713
1.000e-07	65.468	90.631
6.250e-02	65.415	90.742
5.000e-01	64.389	89.833
1.000e+00	51.851	75.809

(d) ViT P-groupings

P-grouping	Test Accuracy	Final Train Accuracy
3.200e+01	52.993	87.847
6.400e+01	52.407	84.940
1.250e-01	49.319	77.788
3.125e-02	49.111	78.160
5.000e-01	48.789	78.229
2.500e-01	48.744	77.060
1.000e-07	48.664	77.181
8.000e+00	48.631	77.302
6.250e-02	48.578	78.662
1.600e+01	48.436	77.230
1.000e+00	47.931	77.659

Table A.4: Average performance of each p-grouping employed in each architecture for training schemes which used momentum and weight decay.

Appendix B

Basic Training Configurations

Below exist the model combinations used in our experiments. Note however that this table **DOES NOT** account for the different p-groupings used in GroupNorm. Keep in mind that each GroupNorm configuration in this table was trained with each appropriate p-grouping. Again, it should be noted that for the ResNet-56, a p-grouping of 64 is not compatible with the standard architecture due to how the ResNet manipulates the dimensions of the features during the throughput process.

Model	Norm	Dataset	Batch Size	Skip Connections	Mom. & Weight Decay
ResNet	BatchNorm	CIFAR-10	16	Yes	Yes
ResNet	BatchNorm	CIFAR-10	16	No	Yes
ResNet	BatchNorm	CIFAR-10	256	Yes	Yes
ResNet	BatchNorm	CIFAR-10	256	No	Yes
ResNet	BatchNorm	CIFAR-100	16	Yes	Yes
ResNet	BatchNorm	CIFAR-100	16	No	Yes
ResNet	BatchNorm	CIFAR-100	256	Yes	Yes
ResNet	BatchNorm	CIFAR-100	256	No	Yes
ResNet	EvoNormBO	CIFAR-10	16	Yes	Yes
ResNet	EvoNormBO	CIFAR-10	16	No	Yes
ResNet	EvoNormBO	CIFAR-10	256	Yes	Yes
ResNet	EvoNormBO	CIFAR-10	256	No	Yes
ResNet	EvoNormBO	CIFAR-100	16	Yes	Yes
ResNet	EvoNormBO	CIFAR-100	16	No	Yes
ResNet	EvoNormBO	CIFAR-100	256	Yes	Yes
ResNet	EvoNormBO	CIFAR-100	256	No	Yes
ResNet	EvoNormSO	CIFAR-10	16	Yes	Yes
ResNet	EvoNormSO	CIFAR-10	16	No	Yes
ResNet	EvoNormSO	CIFAR-10	256	Yes	Yes
ResNet	EvoNormSO	CIFAR-10	256	No	Yes
ResNet	EvoNormSO	CIFAR-100	16	Yes	Yes
ResNet	EvoNormSO	CIFAR-100	16	No	Yes
ResNet	EvoNormSO	CIFAR-100	256	Yes	Yes
ResNet	EvoNormSO	CIFAR-100	256	No	Yes
ResNet	FRN	CIFAR-10	16	Yes	Yes
ResNet	FRN	CIFAR-10	16	No	Yes
ResNet	FRN	CIFAR-10	256	Yes	Yes
ResNet	FRN	CIFAR-10	256	No	Yes
ResNet	FRN	CIFAR-100	16	Yes	Yes
ResNet	FRN	CIFAR-100	16	No	Yes
ResNet	FRN	CIFAR-100	256	Yes	Yes

ResNet	FRN	CIFAR-100	256	No	Yes
ResNet	GroupNorm	CIFAR-10	16	Yes	Yes
ResNet	GroupNorm	CIFAR-10	256	Yes	Yes
ResNet	GroupNorm	CIFAR-10	16	No	Yes
ResNet	GroupNorm	CIFAR-10	256	Yes	Yes
ResNet	GroupNorm	CIFAR-100	16	Yes	Yes
ResNet	GroupNorm	CIFAR-100	16	No	Yes
ResNet	GroupNorm	CIFAR-100	256	Yes	Yes
ResNet	GroupNorm	CIFAR-100	256	No	Yes
ResNet	InstanceNorm	CIFAR-10	16	Yes	Yes
ResNet	InstanceNorm	CIFAR-10	16	No	Yes
ResNet	InstanceNorm	CIFAR-10	256	Yes	Yes
ResNet	InstanceNorm	CIFAR-10	256	No	Yes
ResNet	InstanceNorm	CIFAR-100	16	Yes	Yes
ResNet	InstanceNorm	CIFAR-100	16	No	Yes
ResNet	InstanceNorm	CIFAR-100	256	Yes	Yes
ResNet	InstanceNorm	CIFAR-100	256	No	Yes
ResNet	LayerNorm	CIFAR-10	16	Yes	Yes
ResNet	LayerNorm	CIFAR-10	16	No	Yes
ResNet	LayerNorm	CIFAR-10	256	Yes	Yes
ResNet	LayerNorm	CIFAR-10	256	No	Yes
ResNet	LayerNorm	CIFAR-100	16	Yes	Yes
ResNet	LayerNorm	CIFAR-100	16	No	Yes
ResNet	LayerNorm	CIFAR-100	256	Yes	Yes
ResNet	LayerNorm	CIFAR-100	256	No	Yes
ResNet	Plain	CIFAR-10	16	Yes	Yes
ResNet	Plain	CIFAR-10	16	No	Yes
ResNet	Plain	CIFAR-10	256	Yes	Yes
ResNet	Plain	CIFAR-10	256	No	Yes
ResNet	Plain	CIFAR-100	16	Yes	Yes
ResNet	Plain	CIFAR-100	16	No	Yes
ResNet	Plain	CIFAR-100	256	Yes	Yes

ResNet	Plain	CIFAR-100	256	No	Yes
ResNet	VarianceNorm	CIFAR-10	16	Yes	Yes
ResNet	VarianceNorm	CIFAR-10	16	No	Yes
ResNet	VarianceNorm	CIFAR-10	256	Yes	Yes
ResNet	VarianceNorm	CIFAR-10	256	No	Yes
ResNet	VarianceNorm	CIFAR-100	16	Yes	Yes
ResNet	VarianceNorm	CIFAR-100	16	No	Yes
ResNet	VarianceNorm	CIFAR-100	256	Yes	Yes
ResNet	VarianceNorm	CIFAR-100	256	No	Yes
VGG	BatchNorm	CIFAR-10	16	N/A	Yes
VGG	BatchNorm	CIFAR-10	256	N/A	Yes
VGG	BatchNorm	CIFAR-100	16	N/A	Yes
VGG	BatchNorm	CIFAR-100	256	N/A	Yes
VGG	EvoNormBO	CIFAR-10	16	N/A	Yes
VGG	EvoNormBO	CIFAR-10	256	N/A	Yes
VGG	EvoNormBO	CIFAR-100	16	N/A	Yes
VGG	EvoNormBO	CIFAR-100	256	N/A	Yes
VGG	EvoNormSO	CIFAR-10	16	N/A	Yes
VGG	EvoNormSO	CIFAR-10	256	N/A	Yes
VGG	EvoNormSO	CIFAR-100	16	N/A	Yes
VGG	EvoNormSO	CIFAR-100	256	N/A	Yes
VGG	FRN	CIFAR-10	16	N/A	Yes
VGG	FRN	CIFAR-10	256	N/A	Yes
VGG	FRN	CIFAR-100	16	N/A	Yes
VGG	FRN	CIFAR-100	256	N/A	Yes
VGG	GroupNorm	CIFAR-10	16	N/A	Yes
VGG	GroupNorm	CIFAR-10	256	N/A	Yes
VGG	GroupNorm	CIFAR-100	16	N/A	Yes
VGG	GroupNorm	CIFAR-100	256	N/A	Yes
VGG	InstanceNorm	CIFAR-10	16	N/A	Yes
VGG	InstanceNorm	CIFAR-10	256	N/A	Yes
VGG	InstanceNorm	CIFAR-100	16	N/A	Yes

VGG	InstanceNorm	CIFAR-100	256	N/A	Yes
VGG	LayerNorm	CIFAR-10	16	N/A	Yes
VGG	LayerNorm	CIFAR-10	256	N/A	Yes
VGG	LayerNorm	CIFAR-100	16	N/A	Yes
VGG	LayerNorm	CIFAR-100	256	N/A	Yes
VGG	Plain	CIFAR-10	16	N/A	Yes
VGG	Plain	CIFAR-10	256	N/A	Yes
VGG	Plain	CIFAR-100	16	N/A	Yes
VGG	Plain	CIFAR-100	256	N/A	Yes
VGG	VarianceNorm	CIFAR-10	16	N/A	Yes
VGG	VarianceNorm	CIFAR-10	256	N/A	Yes
VGG	VarianceNorm	CIFAR-100	16	N/A	Yes
VGG	VarianceNorm	CIFAR-100	256	N/A	Yes
Vision Transformer	BatchNorm	CIFAR-10	16	N/A	Yes
Vision Transformer	BatchNorm	CIFAR-10	256	N/A	Yes
Vision Transformer	BatchNorm	CIFAR-100	16	N/A	Yes
Vision Transformer	BatchNorm	CIFAR-100	256	N/A	Yes
Vision Transformer	EvoNormBO	CIFAR-10	16	N/A	Yes
Vision Transformer	EvoNormBO	CIFAR-10	256	N/A	Yes
Vision Transformer	EvoNormBO	CIFAR-100	16	N/A	Yes
Vision Transformer	EvoNormBO	CIFAR-100	256	N/A	Yes
Vision Transformer	EvoNormSO	CIFAR-10	16	N/A	Yes
Vision Transformer	EvoNormSO	CIFAR-10	256	N/A	Yes
Vision Transformer	EvoNormSO	CIFAR-100	16	N/A	Yes
Vision Transformer	EvoNormSO	CIFAR-100	256	N/A	Yes
Vision Transformer	FRN	CIFAR-10	16	N/A	Yes
Vision Transformer	FRN	CIFAR-10	256	N/A	Yes
Vision Transformer	FRN	CIFAR-100	16	N/A	Yes
Vision Transformer	FRN	CIFAR-100	256	N/A	Yes
Vision Transformer	GroupNorm	CIFAR-10	16	N/A	Yes
Vision Transformer	GroupNorm	CIFAR-10	256	N/A	Yes
Vision Transformer	GroupNorm	CIFAR-100	16	N/A	Yes

Vision Transformer	GroupNorm	CIFAR-100	256	N/A	Yes
Vision Transformer	InstanceNorm	CIFAR-10	16	N/A	Yes
Vision Transformer	InstanceNorm	CIFAR-10	256	N/A	Yes
Vision Transformer	InstanceNorm	CIFAR-100	16	N/A	Yes
Vision Transformer	InstanceNorm	CIFAR-100	256	N/A	Yes
Vision Transformer	LayerNorm	CIFAR-10	16	N/A	Yes
Vision Transformer	LayerNorm	CIFAR-10	256	N/A	Yes
Vision Transformer	LayerNorm	CIFAR-100	16	N/A	Yes
Vision Transformer	LayerNorm	CIFAR-100	256	N/A	Yes
Vision Transformer	Plain	CIFAR-10	16	N/A	Yes
Vision Transformer	Plain	CIFAR-10	256	N/A	Yes
Vision Transformer	Plain	CIFAR-100	16	N/A	Yes
Vision Transformer	Plain	CIFAR-100	256	N/A	Yes
Vision Transformer	VarianceNorm	CIFAR-10	16	N/A	Yes
Vision Transformer	VarianceNorm	CIFAR-10	256	N/A	Yes
Vision Transformer	VarianceNorm	CIFAR-100	16	N/A	Yes
Vision Transformer	VarianceNorm	CIFAR-100	256	N/A	Yes
ResNet	BatchNorm	CIFAR-10	16	Yes	No
ResNet	BatchNorm	CIFAR-10	16	No	No
ResNet	BatchNorm	CIFAR-10	256	Yes	No
ResNet	BatchNorm	CIFAR-10	256	No	No
ResNet	BatchNorm	CIFAR-100	16	Yes	No
ResNet	BatchNorm	CIFAR-100	16	No	No
ResNet	BatchNorm	CIFAR-100	256	Yes	No
ResNet	BatchNorm	CIFAR-100	256	No	No
ResNet	EvoNormBO	CIFAR-10	16	Yes	No
ResNet	EvoNormBO	CIFAR-10	16	No	No
ResNet	EvoNormBO	CIFAR-10	256	Yes	No
ResNet	EvoNormBO	CIFAR-10	256	No	No
ResNet	EvoNormBO	CIFAR-100	16	Yes	No
ResNet	EvoNormBO	CIFAR-100	16	No	No
ResNet	EvoNormBO	CIFAR-100	256	Yes	No

ResNet	EvoNormBO	CIFAR-100	256	No	No
ResNet	EvoNormSO	CIFAR-10	16	Yes	No
ResNet	EvoNormSO	CIFAR-10	16	No	No
ResNet	EvoNormSO	CIFAR-10	256	Yes	No
ResNet	EvoNormSO	CIFAR-10	256	No	No
ResNet	EvoNormSO	CIFAR-100	16	Yes	No
ResNet	EvoNormSO	CIFAR-100	16	No	No
ResNet	EvoNormSO	CIFAR-100	256	Yes	No
ResNet	EvoNormSO	CIFAR-100	256	No	No
ResNet	FRN	CIFAR-10	16	Yes	No
ResNet	FRN	CIFAR-10	16	No	No
ResNet	FRN	CIFAR-10	256	Yes	No
ResNet	FRN	CIFAR-10	256	No	No
ResNet	FRN	CIFAR-100	16	Yes	No
ResNet	FRN	CIFAR-100	16	No	No
ResNet	FRN	CIFAR-100	256	Yes	No
ResNet	FRN	CIFAR-100	256	No	No
ResNet	GroupNorm	CIFAR-10	16	Yes	No
ResNet	GroupNorm	CIFAR-10	256	Yes	No
ResNet	GroupNorm	CIFAR-10	16	No	No
ResNet	GroupNorm	CIFAR-10	256	Yes	No
ResNet	GroupNorm	CIFAR-100	16	Yes	No
ResNet	GroupNorm	CIFAR-100	16	No	No
ResNet	GroupNorm	CIFAR-100	256	Yes	No
ResNet	GroupNorm	CIFAR-100	256	No	No
ResNet	InstanceNorm	CIFAR-10	16	Yes	No
ResNet	InstanceNorm	CIFAR-10	16	No	No
ResNet	InstanceNorm	CIFAR-10	256	Yes	No
ResNet	InstanceNorm	CIFAR-10	256	No	No
ResNet	InstanceNorm	CIFAR-100	16	Yes	No
ResNet	InstanceNorm	CIFAR-100	16	No	No
ResNet	InstanceNorm	CIFAR-100	256	Yes	No

ResNet	InstanceNorm	CIFAR-100	256	No	No
ResNet	LayerNorm	CIFAR-10	16	Yes	No
ResNet	LayerNorm	CIFAR-10	16	No	No
ResNet	LayerNorm	CIFAR-10	256	Yes	No
ResNet	LayerNorm	CIFAR-10	256	No	No
ResNet	LayerNorm	CIFAR-100	16	Yes	No
ResNet	LayerNorm	CIFAR-100	16	No	No
ResNet	LayerNorm	CIFAR-100	256	Yes	No
ResNet	LayerNorm	CIFAR-100	256	No	No
ResNet	Plain	CIFAR-10	16	Yes	No
ResNet	Plain	CIFAR-10	16	No	No
ResNet	Plain	CIFAR-10	256	Yes	No
ResNet	Plain	CIFAR-10	256	No	No
ResNet	Plain	CIFAR-100	16	Yes	No
ResNet	Plain	CIFAR-100	16	No	No
ResNet	Plain	CIFAR-100	256	Yes	No
ResNet	Plain	CIFAR-100	256	No	No
ResNet	VarianceNorm	CIFAR-10	16	Yes	No
ResNet	VarianceNorm	CIFAR-10	16	No	No
ResNet	VarianceNorm	CIFAR-10	256	Yes	No
ResNet	VarianceNorm	CIFAR-10	256	No	No
ResNet	VarianceNorm	CIFAR-100	16	Yes	No
ResNet	VarianceNorm	CIFAR-100	16	No	No
ResNet	VarianceNorm	CIFAR-100	256	Yes	No
ResNet	VarianceNorm	CIFAR-100	256	No	No
VGG	BatchNorm	CIFAR-10	16	N/A	No
VGG	BatchNorm	CIFAR-10	256	N/A	No
VGG	BatchNorm	CIFAR-100	16	N/A	No
VGG	BatchNorm	CIFAR-100	256	N/A	No
VGG	EvoNormBO	CIFAR-10	16	N/A	No
VGG	EvoNormBO	CIFAR-10	256	N/A	No
VGG	EvoNormBO	CIFAR-100	16	N/A	No

VGG	EvoNormBO	CIFAR-100	256	N/A	No
VGG	EvoNormSO	CIFAR-10	16	N/A	No
VGG	EvoNormSO	CIFAR-10	256	N/A	No
VGG	EvoNormSO	CIFAR-100	16	N/A	No
VGG	EvoNormSO	CIFAR-100	256	N/A	No
VGG	FRN	CIFAR-10	16	N/A	No
VGG	FRN	CIFAR-10	256	N/A	No
VGG	FRN	CIFAR-100	16	N/A	No
VGG	FRN	CIFAR-100	256	N/A	No
VGG	GroupNorm	CIFAR-10	16	N/A	No
VGG	GroupNorm	CIFAR-10	256	N/A	No
VGG	GroupNorm	CIFAR-100	16	N/A	No
VGG	GroupNorm	CIFAR-100	256	N/A	No
VGG	InstanceNorm	CIFAR-10	16	N/A	No
VGG	InstanceNorm	CIFAR-10	256	N/A	No
VGG	InstanceNorm	CIFAR-100	16	N/A	No
VGG	InstanceNorm	CIFAR-100	256	N/A	No
VGG	LayerNorm	CIFAR-10	16	N/A	No
VGG	LayerNorm	CIFAR-10	256	N/A	No
VGG	LayerNorm	CIFAR-100	16	N/A	No
VGG	LayerNorm	CIFAR-100	256	N/A	No
VGG	Plain	CIFAR-10	16	N/A	No
VGG	Plain	CIFAR-10	256	N/A	No
VGG	Plain	CIFAR-100	16	N/A	No
VGG	Plain	CIFAR-100	256	N/A	No
VGG	VarianceNorm	CIFAR-10	16	N/A	No
VGG	VarianceNorm	CIFAR-10	256	N/A	No
VGG	VarianceNorm	CIFAR-100	16	N/A	No
VGG	VarianceNorm	CIFAR-100	256	N/A	No
Vision Transformer	BatchNorm	CIFAR-10	16	N/A	No
Vision Transformer	BatchNorm	CIFAR-10	256	N/A	No
Vision Transformer	BatchNorm	CIFAR-100	16	N/A	No

Vision Transformer	BatchNorm	CIFAR-100	256	N/A	No
Vision Transformer	EvoNormBO	CIFAR-10	16	N/A	No
Vision Transformer	EvoNormBO	CIFAR-10	256	N/A	No
Vision Transformer	EvoNormBO	CIFAR-100	16	N/A	No
Vision Transformer	EvoNormBO	CIFAR-100	256	N/A	No
Vision Transformer	EvoNormSO	CIFAR-10	16	N/A	No
Vision Transformer	EvoNormSO	CIFAR-10	256	N/A	No
Vision Transformer	EvoNormSO	CIFAR-100	16	N/A	No
Vision Transformer	EvoNormSO	CIFAR-100	256	N/A	No
Vision Transformer	FRN	CIFAR-10	16	N/A	No
Vision Transformer	FRN	CIFAR-10	256	N/A	No
Vision Transformer	FRN	CIFAR-100	16	N/A	No
Vision Transformer	FRN	CIFAR-100	256	N/A	No
Vision Transformer	GroupNorm	CIFAR-10	16	N/A	No
Vision Transformer	GroupNorm	CIFAR-10	256	N/A	No
Vision Transformer	GroupNorm	CIFAR-100	16	N/A	No
Vision Transformer	GroupNorm	CIFAR-100	256	N/A	No
Vision Transformer	InstanceNorm	CIFAR-10	16	N/A	No
Vision Transformer	InstanceNorm	CIFAR-10	256	N/A	No
Vision Transformer	InstanceNorm	CIFAR-100	16	N/A	No
Vision Transformer	InstanceNorm	CIFAR-100	256	N/A	No
Vision Transformer	LayerNorm	CIFAR-10	16	N/A	No
Vision Transformer	LayerNorm	CIFAR-10	256	N/A	No
Vision Transformer	LayerNorm	CIFAR-100	16	N/A	No
Vision Transformer	LayerNorm	CIFAR-100	256	N/A	No
Vision Transformer	Plain	CIFAR-10	16	N/A	No
Vision Transformer	Plain	CIFAR-10	256	N/A	No
Vision Transformer	Plain	CIFAR-100	16	N/A	No
Vision Transformer	Plain	CIFAR-100	256	N/A	No
Vision Transformer	VarianceNorm	CIFAR-10	16	N/A	No
Vision Transformer	VarianceNorm	CIFAR-10	256	N/A	No
Vision Transformer	VarianceNorm	CIFAR-100	16	N/A	No

Vision Transformer	VarianceNorm	CIFAR-100	256	N/A	No