Exploring the optimization process in Neurally Reparameterized Topology Optimization

Exploring the optimization process in Neurally Reparameterized Topology Optimization

by

Suryanarayanan M S

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday November 9, 2021 at 03:00 PM.

Student number:	5052203	
Daily supervisor:	Gawel Kus	
Project duration:	January, 2021 – October, 2021	
Thesis committee:	Dr. Miguel A. Bessa,	TU Delft, supervisor
	Prof. dr. ir. A. (Fred) van Keulen,	TU Delft
	Dr. Alejandro M. Aragón,	TU Delft
	Dr. Deepesh Toshniwal,	TU Delft

An electronic version of this thesis is available at
http://repository.tudelft.nl/.

Keywords: Optimization, neural networks, topology, loss landscape

Front & Back: Evolution of design for a structure optimized in neural space.

Copyright © 2021 by Suryanarayanan M S

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Poets say science takes away from the beauty of the stars - mere globs of gas atoms. I, too, can see the stars on a desert night, and feel them. But do I see less or more?

Richard P. Feynman

Summary

Inverse design with topology optimization has followed the same computational graph for decades. The unknown material density is distributed within a domain, a computational analysis predicts the response of that design and its derivative with respect to the unknown, and this information is used by a chosen gradient-based optimization algorithm to find the next design iteration until it reaches an optimum (local or global). Recently, however, a counter-intuitive strategy was proposed which augments the computational graph by including a neural network in between the response prediction (computational analysis) and the generation of a new design (image). This shifts the optimization problem from its original space to the weight space of the neural network. Yet, this indirect optimization process was shown elsewhere to outperform conventional topology optimization for a large number of structural compliance problems – at least when choosing a particular convolutional neural network (part of a U-Net) and a particular optimizer (L-BFGS). This investigation provides quantitative and qualitative arguments that justify why these choices are successful, concluding that the line-search component of L-BFGS is key to traversing the reparameterized objective (loss) landscape and quickly reaching good solutions in "flat" regions of the landscape. Importantly, these topology optimization problems are not stochastic which make them different from the majority of conventional deep learning applications, favoring the use of line-search. Similarly, although to lesser extent, the approximation of the Hessian provided by L-BFGS helps moving more effectively within the flat regions by rescaling the gradients, the quality of the approximation is less relevant. Together with the deep image prior effect associated to deep learning, these arguments explain the early success of the neural reparameterization strategy in topology optimization, in spite of the non-convex objective landscape distortion that they introduce even for landscapes that were originally convex.

Contents

Summary **vii**

1 Introduction **1**

2 Literature review **5**
2.1 Machine Learning in Topology Optimization 5
2.1.1 Neural reparameterization of TO 6
2.2 Comparing neurally reparameterized TO and DL 10
2.2.1 Notation and Terminology 12
2.3 Optimizers 13
2.3.1 Gradient Descent (GD) 14
2.3.2 Limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) 15
2.3.3 Usage in DL Literature 17
2.4 Optimization landscapes 20
2.4.1 Visualization tools 22
2.4.2 Analysis tools 25

3 Understanding neural optimization of topology **31**

4 Discussion **47**
4.1 Limitations and future research 48

References **51**

A Supplementary information for Chapter - 2 **57**
A.1 BFGS algorithm details 57
A.2 Observed empirical properties of neural landscapes . . . 58
A.2.1 Quasi-convexity 58
A.2.2 Mode connectivity and predictability 60

B Supplementary information for chapter-3 **63**
B.1 Computational graph details 63
B.1.1 Basic framework 63
B.1.2 Twice differentiable graph 63
B.1.3 TO problems 64
B.2 Optimizer details 64
B.2.1 L-BFGS 64
B.2.2 Gradient Descent (GD) 64
B.2.3 Hessian Descent (HD) 64
B.2.4 Gradient Descent with line-search (GD_LS) 64

B.3 Hessian EDS for the loss function 65
B.4 Computing the approximate Hessian's spectra 65
B.5 Hyper-parameters for Hessian analysis 67
B.6 Test codes for appendix 68
B.7 Hyper-parameter optimization 68

C Supplementary information for chapter-4 73
C.1 Loss curves 73
C.2 Eigenvalue Density Spectrum 74
C.3 Linear interpolation 74
C.4 2D projections 75
C.5 Distance and gradient norm 76

1

Introduction

Topology optimization (TO) is a sub-field of structural optimization where a limited amount of material is to be distributed in a design region, subjected to certain forces and supports, so as to obtain the desired structural properties. Conventionally, a density based TO is solved by first discretizing the design domain into finite elements and then determining iteratively which elements should hold material, so as to minimize (or maximize) a certain property like compliance, fracture toughness etc [1]. In short, this process progressively optimizes a structure, parameterized as the density values of FEM elements, using a standard optimization algorithm. The schematic for this conventional method is shown in Figure 1.1.

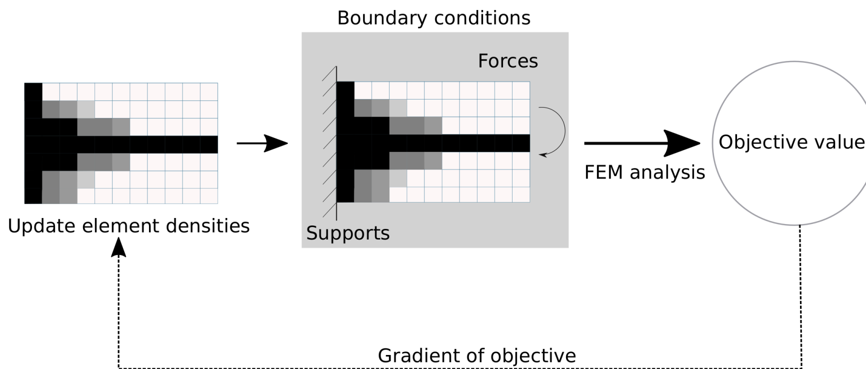


Figure 1.1: Schematic for a conventional density based TO. The darker regions correspond to densities near 1, indicating the presence of material while the white or lighter regions are devoid of material.

In recent years, techniques from the field of Machine Learning (ML) and especially Deep Learning (DL) are increasingly being applied to various scientific domains [2].

Although there have been various attempts to accelerate or improve TO using such techniques (as reviewed in section 2.1 of chapter 2), most of these methods were similar to conventional supervised Machine Learning (ML). The general procedure was to form a dataset by running TO simulations and use this to train a ML/DL model. Once trained, these models are able to predict the optimized topology without running the expensive TO simulation again. The various works in this category differ mainly in the inputs chosen to train the model or on the model algorithm itself.

Recent papers by Hoyer et al [3], and later by Chandrasekhar et al [4], have shifted away from this paradigm and have introduced the concept of *neural reparameterization* to the field of TO. Both these papers ponder the question of shifting the optimization process from the conventional FEM grid based space to the weights of a deep convolutional neural network (CNN). Since the generated structures in TO are parameterized by FEM elements with values between 0 and 1, they share similarities to natural images which are parameterized by pixel values, as illustrated in Figure 1.2. The expectation was that the neural network's expressiveness and especially their success in tasks relating to images would make them ideal for reparameterizing TO problems as well. Hoyer et al [3] managed to test this hypothesis on 116 different problems for compliance TO against conventional baselines that were limited to the pixel space. The hypothesis was confirmed and the newly proposed method was found to be either comparable or better than the baselines on all the problems. The success behind this neural reparameterization strategy is thought to be due to the inductive biases embedded in the network's architecture [3, 6]. But the answer to how these "deep image priors" affect the geometry of the landscape and influence the optimization process is still being studied in the DL literature. The framework introduced by Hoyer et al [3] is an excellent sandbox to investigate these effects. As a step towards these big questions, this thesis will investigate the optimization process in the neural weight space using this framework.

At the beginning of the thesis, we were inspired by a technical detail that was mentioned in the paper [3]. The authors had used the L-BFGS optimizer to update the weights of the CNN because it was found to outperform Stochastic Gradient Descent (SGD) on all the tested problems. L-BFGS was known to employ curvature information and is meant to be effective in convex or strongly convex settings while SGD enjoyed the status of being the optimizer of choice for most DL applications [7]. This made us wonder whether the curvature information encoded in the Hessian matrix of the objective function and the properties of the underlying landscape would provide the crucial details behind the success of neural reparameterization. Also it is understood that the synergy between the optimizer and the landscape is crucial for the success of any optimization problem. With these insights, we decided to investigate the optimization process by exploring the differences between these two optimizers, as they travel through several landscapes in compliance topology optimization.

As neural reparameterization involves the use of deep neural networks, it was nec-

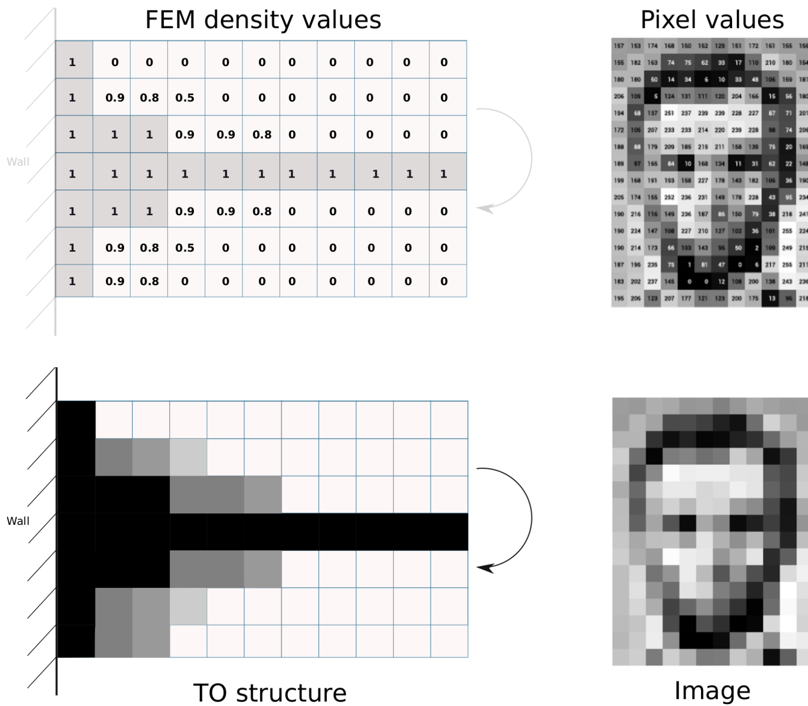


Figure 1.2: Similarity between the structure generated at an intermediate TO step and a grey-scale image. Part of the image was taken from [5].

essary to review the DL literature to understand the characteristics of the loss landscapes of these models. Further, a detailed study into the algorithms of both L-BFGS and SGD was also important. The information regarding both these aspects has been condensed and presented in chapter 2. After the literature survey, various tools and techniques for loss landscape exploration and methods to decompose the L-BFGS optimizer were developed applied to the current settings. The relevant results of the experiments conducted are presented in chapter 3, which can be read as a standalone chapter. Finally, a extended discussion on the results and the limitations of the thesis are brought together and the whole picture is presented in chapter 4.

2

Literature review

The primary goal of the literature survey is to search for tools and techniques to analyze the optimization process based on the similarities this method shares with other Machine Learning/ Deep Learning (DL) approaches. In order to achieve this goal, it is necessary to delineate the optimization into its two components; objective landscapes and optimizers and to study each in detail. Thus, this chapter will first present the problem formulation of applying neural reparameterization to the field of compliance Topology optimization. This is followed by a discourse into the algorithms of the gradient-based optimizers L-BFGS and SGD and their typical usage in DL applications. Finally, the chapter will review the tools and techniques used in DL literature to study neural loss landscapes.

2.1. Machine Learning in Topology Optimization

As mentioned in chapter 1, there have been various attempts to bring the benefits of Machine Learning (ML) and Deep Learning (DL) to the field of Topology Optimization (TO). Broadly, they can be classified into 2 categories; ones that use a dataset explicitly and others that utilize data implicitly. The former category is very similar or even equivalent to the supervised learning setting, where a dataset formed by running TO simulations is used to train a ML/DL model. Once trained, these models are able to predict the optimized topologies without running the expensive TO simulation again. The various works in this category differ either in the inputs chosen to train the model or on the model algorithm itself. For example, Sosnovik et al [8] and Zhang et al [9] both use the dataset obtained by performing at least one TO iteration on a Convolutional Neural Network (CNN) architecture. But the main difference is that while the input features in [8] are the intermediate TO design and the change in the design from the previous TO iteration, the inputs used in [9] are the nodal displacements and strains. Some related methods utilize forms of dimensionalty reduction to transform the inputs to the model, as in [10–12]. Other

approaches may utilize different ML/DL models such as Variational AutoEncoders (VAEs) [13], CNNs or Generative Adverserial Networks (GANs) [14] for training the TO dataset. The major drawbacks of these methods are the requirements for a computationally expensive training dataset and extensive feature engineering of the inputs. Further, since the datasets are seldom exhaustive, the generalization capabilities and the exactness of the physics learned are both limited.

The second category of methods are aimed at providing a complementary approach to the explicit-data methods. Two recent papers, one by Hoyer et al [3] and other by Chandrasekhar et al [4] are relevant. Both are conceptually similar but differ in their implementation. Both these methods allow the incorporation of exact physics, do not require any training data and perform optimization in a neural weight space. The main drawback of [4] is that it is difficult to attribute a single factor to its success. For example, the authors mention that they change the penalization factor and regularization constant alongside the optimization process. Further, the neural network is trained using Adam optimizer while the baseline chosen is the conventional Optimality Criteria (OC). From the results of [3], it can be seen that method of moving asymptotes (MMA) is a better baseline than OC for many problems. Also, Hoyer et al [3] delineates the effect of the optimizer from their approach by providing a baseline method that utilizes the same optimizer as used for training the neural network. Considering these factors, the Hoyer approach [3] is chosen for further study and is explained in detail in the next section even though the tools and techniques reviewed in this survey are applicable for both the explicit and implicit data methods.

2.1.1. Neural reparameterization of TO

The content of the paper [3] can be roughly decomposed into 3 separate concepts whose understanding is crucial to proceed further. The first of these is the term "*Neural*" which is meant to represent neural networks (NNs). NNs are prominent in the fields of ML and especially DL and are often used as surrogate models because of their ability to approximate any function [7]. The next term of interest is "*Optimization*". The field of optimization pervades numerous applications. Wherever there are problems that require the maximization or minimization of some function, this term is significant. Within the context of the paper, the field of topology optimization (TO) was studied. Finally, the term "*Reparameterization*", in simple terms, means a change of basis. It is a reformulation of a given problem from one set of coordinate axes to another. Now, we will dive into each of these concepts briefly.

TO using modified SIMP approach

One of the most successful methods to perform TO when there is easy access to gradient information is by using the Solid Isotropic Material with Penalization (SIMP) approach [15]. At each iteration of the optimization process, the current material distribution layout forms a structure and in [3], the compliance of this structure was calculated using an FEM solver with a modified SIMP approach [1]. Instead of treating the design variables (elements of the FEM grid) as having discrete bi-

nary values, which does not permit gradient-based optimization, the SIMP approach treats these as continuous densities that can take any value between 0 and 1. The modified SIMP approach assigns a Young's modulus (E_e) for each of the elements based on their density value (x_e) using the formula

$$E_e(x_e) = E_{min} + x_e^p(E_0 - E_{min}) \tag{2.1}$$

where E_0 is the Young's modulus of the material under consideration and E_{min} is a very small stiffness value assigned to the void areas to make sure that the inverse of the stiffness matrix is defined. The latter addition makes the modified approach differ from the classical SIMP. The factor p is the penalization factor that causes convergence to 0-1 solutions by penalizing intermediate density values. The choice of p and the objective function is crucial in the optimization process. For example, it has been shown that for the original SIMP approach with $p = 1$ and a compliance objective, the function is convex with a unique solution. But for other values of p, no such guarantees exist and the general compliance TO problem can be non-convex [15]. Using this modified SIMP approach, the problem in [3] is a compliance TO with $p = 3$ formulated as

$$\begin{aligned} \min_{\mathbf{x}} : g(\mathbf{x}) = \mathbf{U}^\mathsf{T}\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} E_e(x_e)\mathbf{u}_e^\mathsf{T}\mathbf{k}_o\mathbf{u}_e \\ \text{subject to} : \mathbf{K}\mathbf{U} = \mathbf{F} \\ V(\mathbf{x})/V_0 = f \\ 0 \leq x_e \leq 1 \end{aligned} \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^N$ is the vector of design variables, N is the number of elements, $\mathbf{U}$ is the global displacement vector, $\mathbf{K}$ is the global stiffness matrix and $\mathbf{F}$ is the force vector. Similarly, $\mathbf{u_e}$ and $\mathbf{k_0}$ are the element-wise displacement vector and stiffness matrix respectively. The volume constraint is applied by setting the volume occupied by the elements to be equal to the fraction f of the total domain volume V_0. This formulation forms the physics part of the entire computational graph. This entire part is treated as a black-box for the purposes of this report and in brief, the physics part is just a function that assigns a compliance value to a given design of a structure i.e. $g : \mathbb{R}^{m \times n} \longrightarrow \mathbb{R}^+$, where m and n are the number of elements in the x and y directions respectively.

Reparameterization in optimization

Reparameterization is a technique that is used very often in solving complex problems. In essence, it is a way to reformulate the same problem using a different parameterization scheme. The simplest case is a change of coordinates from Cartesian to Spherical or Cylindrical coordinates for certain engineering problems. From experience, this can simplify the problem greatly. Other forms of reparameterization that are common in the scientific field are Fourier transforms, Laplace transforms, Reciprocal space etc. The aim is to make the problem more amenable to solve or visualize.

The use of this idea in [3] was inspired from [16], where the authors showed the advantages of changing the parameterizations of an image. In [16], the authors demonstrated that several tasks like feature visualization and neural art use images as an input or output and can benefit greatly if the image's natural pixel parameterization is replaced by a different and differentiable parameterization scheme. One of the cases where they demonstrated this effect was for style transfer applications, where the painting *style* of a particular image is to be transferred onto the *content* of another image to create a combination. This is normally achieved using a CNN which generates the optimized image and the authors remarked that this process has only been achieved on VGG architectures. The authors could achieve style transfer on non-VGG networks by reparameterizing the optimized image onto a Fourier space and performing the optimization process there.

The authors also commented on why such parameterization strategies are important in optimization. Firstly, some reparameterizations can make the optimization process easier by performing a sort of preconditioning of the objective landscape. This achieves similar effects to the case of "scaling" the gradients as discussed in subsection 2.3.2. Secondly, the basins of attraction of various minima are changed. Thirdly, some parameterizations cannot cover the entire spectrum of possibilities and this together with the second fact can help the optimization proceed to newer and better minima. Finally, new parameterizations may offer more degrees of freedom because each parameterization scheme has a distinct representation of the objective. For example, a NN trained on images will have the representation of the images in its parameters. These parameters represent not the pixel values of the images but a very different rich internal representation.

These facts are relevant in the case of TO because the structure obtained at each step of the iteration can be considered as an image, with each FEM element as a pixel with a value between 0 and 1. If the image is thus reparameterized into another space, this could potentially result in better designs and a faster optimization process. The schematic of such a reparameterization is shown in Figure 2.1.

Using Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have had great success in fields such as computer vision where the use of images are important. Natural images possess certain qualities or invariances which "resonate" with these networks. In [6], the authors showed that they can perform image restoration i.e. obtain the true image from a degraded one, using no other information but the degraded image $\mathbf{X_0}$ and a CNN. The image priors embedded in the architecture of the network help the optimization process reach the true image $\mathbf{X}$. The methodology followed in [6] is as follows: If $\vec{\theta}$ is the parameter vector of the network and $\mathbf{z}$ is a constant random vector that acts as the input, then the generated image is $h_{\vec{\theta}}(\mathbf{z})$, where $h : \mathbf{z} \longrightarrow \mathbf{X}$ is the function represented by the CNN. They then iteratively change the parameters of the network such that $\mathbf{X}$ and $\mathbf{X_0}$ are similar. Since the structures obtained in TO have the properties of images, a similar strategy can go hand in hand with

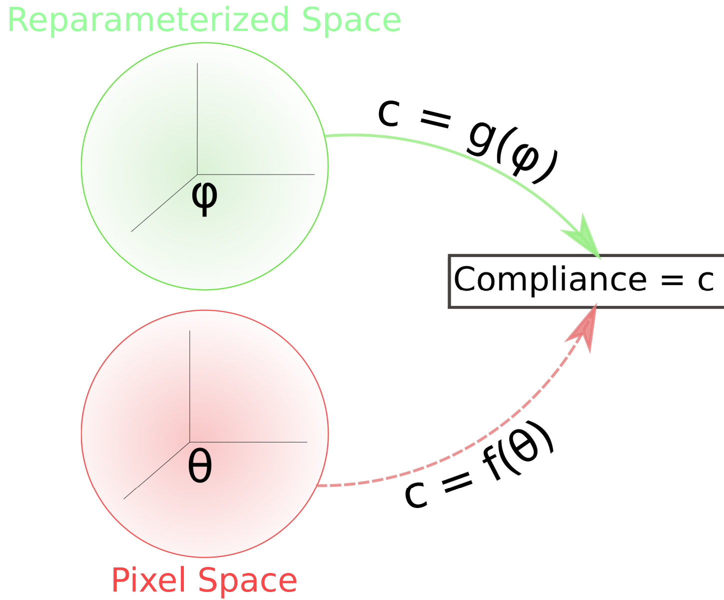


Figure 2.1: The concept of reparameterization. Conventionally in TO, the designs in the Pixel Space are mapped to a compliance value using the function *f* parameterized by $\theta \in \mathbb{R}^N$, where N is the number of pixels. When a new parameterization scheme is chosen, the representation in the new space is mapped onto the compliance value using the function *g* with parameters $\phi \in \mathbb{R}^x$, where x is the number of parameters of the reparameterized space.

generating realistic structures that can maintain spatial features.

Performance of neural reparameterization

The three concepts described above form the ingredients for the approach undertaken by Hoyer et al [3]. The authors managed to make use of the 3 concepts to make better designs with lower objective values consistently. They first implemented the TO physics as a differentiable model using Autograd numpy [17]. Then they reparameterized the TO problem from the Pixel space to the weight space of a CNN with an architecture inspired from [6]. This meant that instead of optimizing the densities on a 2D pixel grid by directly varying the pixel values, they nudged the weights and biases of the CNN using a gradient-based optimizer to find the best design. The output of the CNN was transformed to respect the volume constraint and this resulted in the image of a structure, which could then be analyzed using the TO physics to obtain the compliance value. Since the entire computational graph is differentiable, the gradients can flow from the loss value to the weights and biases of the CNN. This schematic is shown in Figure 2.2.

They formulated 116 TO tasks that were either different problems or the same problem with variations in the mesh size or the volume fraction (f). The CNN repa-

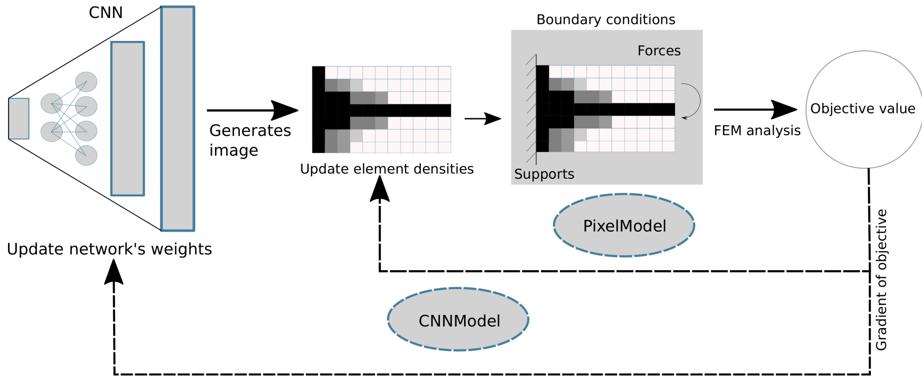


Figure 2.2: A Simplified schematic of the computational graph adopted in [3]. The PixelModel is the conventional way of solving the TO problem while the CNNModel is the proposed method.

rameterized model (CNNModel) was contrasted with the conventional pixel space TO (PixelModel) on these tasks over 101 random seed initializations. In order to serve as baselines, the PixelModels were optimized using the 3 optimizers MMA, OC and L-BFGS while the CNNModel was optimized using L-BFGS. It was found that the CNNModels outperformed the baseline methods for both the best-of-ensemble case as well as for a typical sample, especially for the larger problems.

2.2. Comparing neurally reparameterized TO and DL

The method of neural reparameterization employed in [3] has similarities with the supervised Deep Learning (DL) problem. When decomposing the field of ML/DL, the twin aspects of optimization and generalization are of utmost importance. In DL applications, what we are trying to achieve is to train the neural network (NN) using a dataset with the hope that the network learns the underlying distribution. If it can faithfully learn the underlying function, then the network will perform well for unseen data. If the underlying distribution is $\mathcal{D}$, then the NN should try to model this function from the input space $\mathbf{x}$ to the space of outputs $\mathbf{y}$, as accurately as possible. What we are in fact trying to achieve is the minimization of a quantity called the *expected loss*, which would reduce the discrepancy over all possible inputs. If we could do this, the network would generalize well. Achieving this generalization capability is the aim in most DL applications and minimizing the *expected loss* is the objective function. However, the challenging part is that since we only have a finite number of samples from this distribution, we do not have direct access to the objective function. Instead, what is often done is to minimize the *empirical (or the observed) loss* over a subset $\tilde{\mathcal{D}}$ of samples considered, in a process called training [18]. It is to be noted that the *empirical loss* is a proxy objective and is not the true aim. This training procedure involves minimizing a function and is thus an optimization process.

The particular case of optimization that we are interested in is finding the weights and biases of the NN that would minimize the empirical loss. This utilizes the training dataset and the performance is measured against a chosen loss function. Formally, if the NN represents the function h, then for a given input $\mathbf{x}_i$, the network's scalar output is $y_i = h_{\vec{\theta}}(\mathbf{x}_i)$. Let $\mathcal{L}$ be the loss function that measures the deviation of y_i from a true label $\hat{y}_i$. If $\vec{\theta}$ is the vector of weights and biases of the network, then for $'m'$ examples from a dataset, we have the empirical risk as

$$E_{\text{risk}} = \frac{1}{m}\sum_{i}^{m} \mathcal{L}(h(\vec{x}_i; \vec{\theta}), \hat{y}_i) \tag{2.3}$$

Then, the optimization task becomes

$$\vec{\theta}^* = argmin_{\vec{\theta}}(E_{\text{risk}}) \tag{2.4}$$

where $\vec{\theta}^*$ is the vector containing the optimal weights and biases that minimize the empirical risk.

In the context of [3], the loss function $\mathcal{L}$ is the TO physics and the dataset is generated implicitly. Since the aim is to only find the structure with the lowest objective value, the topic of optimization is the one that is relevant. As the entire optimization process has to be repeated for each TO problem, there is no aspect of generalization. Also, as we have complete access to the objective function, there is no stochasticity involved in the estimation of the gradient. Even though the ultimate aims are different, studying the optimization process in DL applications can provide valuable information that can be useful to the current setting.

Optimization process

Any optimization process consists of two components. The first of these is the optimization landscape or loss landscape as is commonly known in DL literature. In DL settings, once the architecture of the NN, the dataset and the loss function is chosen, this landscape is already fixed [19]. And depending on these factors, the landscape may be conducive to an easier optimization process or it might hinder it. The second entity is the optimizer which traverses this landscape to find the solution. The optimizer-landscape synergy is what enables successful optimization and understanding this in the case of reparameterized TO problems is the main challenge. We will delve briefly into each of these entities.

The algorithms used for finding the optimal weights of neural networks are usually iterative gradient-based optimizers. The gradients of the loss function with respect to the weights and biases of the network are easily calculated because this is integrated into modern ML/DL frameworks. Thus, this literature survey will only be concerned with such optimizers. Further, the optimizer that was used in [3] was L-BFGS. The standard optimizer usually used in ML/DL problems is Stochastic Gradient Descent (SGD) [7]. Since it was mentioned that SGD did not yield good results for the TO problems in comparison to L-BFGS, we will try to focus on these two optimizers in greater detail, with the hope that the differences between these algorithms

might give us valuable information about the optimization process. Also, another reason to study SGD is because most of the literature concerning loss landscapes in ML/DL uses SGD or some variation of it to a great extend [18].

2

The loss landscapes of DL applications are being extensively studied from multiple perspectives [20]. These landscapes, even though they are non-convex, are thought to be different from other complex non-convex landscapes such as that of molecular conformations which are riddled with high error local minima [21]. In such landscapes, gradient-based optimizers are not expected to be highly effective because they get stuck in such sub-optimal minima. This qualitative difference of neural network (NN) loss landscapes is empirically inferred from the ease of optimization, verified with simple optimizers like SGD that can reach global minima of zero training loss on multiple tasks [22]. The loss landscapes in Hoyer et al [3] may or may not share similarities with common DL landscapes. The main difference is that, in most DL cases, the loss function chosen is convex (e.g. Mean Square Error loss). This makes it easier to understand whether the global minimum or a minima of similar quality has been reached in the training process (whether these minima generalize well is a question for the generalization field). In the models of [3] however, the loss function originates from the physics part of the Topology Optimization and is not known to be convex. This makes it hard to judge the quality of the final converged solution obtained from applying any optimizer. All one can hope for is a comparison of the converged solutions of different optimizers or through the analysis of the loss landscapes.

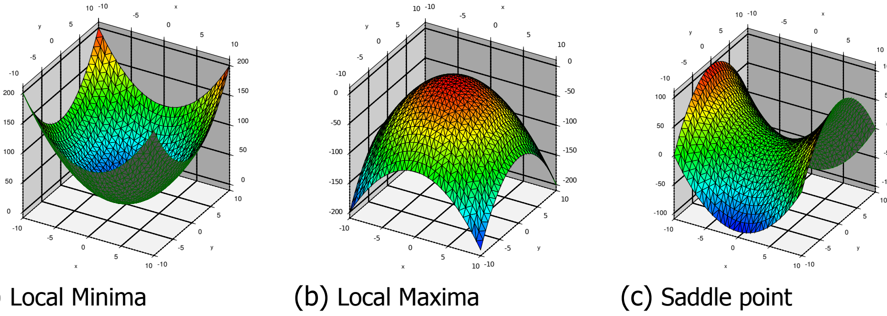
2.2.1. Notation and Terminology

If the elements of the matrix are real numbers and it has m rows and n columns, then the matrix is denoted as $\mathbf{M} \in \mathbb{R}^{m \times n}$ and each of its elements as M_{ij}, $1 \leq i \leq m, 1 \leq j \leq n$. An important matrix is the identity matrix $\mathbf{I}$, with its diagonal elements as one and all other elements as zeros. It can be represented as $\mathbf{I} \in \mathbb{R}^{n \times n}$, $I_{ii} = 1 \quad \& \quad I_{ij} = 0$. A vector of dimension n is denoted either as $\mathbf{v}$ or $\vec{\beta} \in \mathbb{R}^n$. Finally, $diag(\lambda_1, \cdots, \lambda_n)$ will represent the $n \times n$ matrix whose diagonal elements are $\lambda_1, \cdots, \lambda_n$ respectively.

A way to measure the magnitude of a vector or matrix is through the use of *norms*. There are many different kinds of norms used in practice but one of the most common ones is the Frobenius norm for matrices and the L_2 norm for vectors. The L_2 norm of a vector $\mathbf{v}$ is defined as $|\mathbf{v}|_2 = \sum_i v_i^2$ and represents the Euclidean distance from the origin to the point in the high-D space which is represented by the vector. For a matrix, it is defined similarly as the sum of the squares of its elements and thus, $||\mathbf{M}|| = \sum_i \sum_j M_{ij}^2$. The usage of the term *norm* will be synonymous with both the Frobenius norm and the L_2 norm, the exact one being clear from the context.

Gradient vector

For the optimization task represented by Equation 2.4, the gradient of the function $E_{\text{risk}} = f$ with respect to $\vec{\theta}$ is a important quantity. It is a vector with the same



(a) Local Minima (b) Local Maxima (c) Saddle point

Figure 2.3: Different kinds of critical points.

dimensions as $\vec{\theta}$ and is denoted by $\vec{\nabla} f = \frac{\partial f}{\partial \vec{\theta}}$. Geometrically, each component of this vector denotes the slope of the tangent to the surface of the function f, at a given point in a given coordinate direction. In the fields of ML and DL, these gradients are calculated automatically using the technique of back propagation (usually via automatic differentiation) and is implemented in most common DL libraries.

Hessian matrix

The gradient, discussed above, can be seen as the extension of first partial derivatives to the case when a function has a multi-variable input and scalar output. Mathematically, if a function is denoted by $g : \mathbb{R}^m \to \mathbb{R}$, the gradient is an m dimensional vector. In a similar fashion, the Hessian is the natural extension of the second-order partial derivatives. Thus, the Hessian is a matrix of the form $\mathbf{H} = \nabla^2 f \in \mathbb{R}^{m \times m}$, $H_{ij} = \frac{\partial^2 f}{\partial \theta_i \partial \theta_j}$. From the property of second-order derivatives ($\frac{\partial^2 f}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 f}{\partial \theta_j \partial \theta_i}$), the Hessian is a symmetric matrix with $\mathbf{H} = \mathbf{H}^\mathsf{T}$. Geometrically, the Hessian contains the curvature information of the surface of the function. The eigenvalues of the Hessian matrix, where the number of eigenvalues is equal to the number of parameters, are the curvatures along the corresponding eigenvector directions. If all the eigenvalues are positive (negative) at a critical point, then the curvature along all directions is positive (negative) and is thus a local minima (maxima). If some of the eigenvalues are positive and some are negative, then it is saddle point. This is demonstrated for a simple 2D function in Figure 2.3.

2.3. Optimizers

Gradient-based optimizers utilize the gradient information about the objective function to iteratively take steps towards a critical point. At the critical point, since the gradient is zero, the optimizer stops. There are a number of gradient-based optimizers but they can be roughly classified into 2 categories; *first-order* and *second-order* optimizers [7]. The members of the former class utilize only the gradient information while the members of the latter class utilize some extra information, usually the curvature information through the Hessian matrix. We can imagine that the first-order methods, being blind to the local curvature, might take steps that can

2

actually cause an increase in the function value. The most prominent methods in this class are the gradient descent (GD) algorithms and their derivatives. The basic GD algorithm and its more practical version called Stochastic GD (SGD) will be discussed in the next section. Mathematically, we can model the objective function as a local quadratic function using the Taylor's theorem as:

$$f(\mathbf{x_k} + \alpha \mathbf{p}) = f(\mathbf{x_k}) + \alpha \mathbf{p}^T \vec{\nabla} f_k + \frac{1}{2}\alpha^2 \mathbf{p}^T \vec{\nabla}^2 f(\mathbf{x_k} + t\mathbf{p})\mathbf{p}, \quad t \in (0, \alpha) \tag{2.5}$$

where α is a positive scalar called the step-size, $\mathbf{p}$ is the step direction, $\mathbf{x}_k$ represents k^{th} iterate along the optimization trajectory (a point in the objective function landscape) and $\vec{\nabla} f_k$ is the gradient at $\mathbf{x}_k$ [23]. All the optimizers try to find the minimizer of this quadratic model. The general search direction is given by $\mathbf{p} = \mathbf{B}^{-1}\vec{\nabla} f$, where the matrix $\mathbf{B}$ is a preconditioner for the gradient vector. When $\mathbf{B} = \mathbf{H}$, the Hessian of the objective function w.r.t the model's parameters, this becomes the Newton's method and is a second-order method. Since the Hessian of large neural networks is impossible to store, this method is practically not possible for modern DL. On the other end of the spectrum are the Gradient Descent (GD) methods where $\mathbf{B} = \mathbf{I}$. These methods will be explained in the upcoming section. Between these two extremes are Quasi-Newton methods that form an approximation to the Hessian as the $\mathbf{B}$ matrix. They are powerful methods with super-linear convergence rates for convex cases and one method of this class will be discussed in subsection 2.3.2.

2.3.1. Gradient Descent (GD)

Also known as steepest GD, this is the simplest of gradient-based algorithms and take steps in the opposite direction of the gradient vector. Then, for GD, $\mathbf{p}_k = -\vec{\nabla} f_k$ and the iteration rule is simple.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_k \tag{2.6}$$

where α is called the step size and is a tunable hyper-parameter that is to be adjusted based on the problem at hand. Large step sizes allow the algorithm to explore larger distances in the weight space and may even prevent it from converging to sub-optimal local minima. In the opposite case of having smaller step sizes, the algorithm becomes exploitative of the local region and can converge faster to a nearby minima, as shown in Figure 2.4. Thus, the choice of this hyper-parameter is of at most importance and is usually tuned either through a grid search or random search to a suitable value before commencing the full training [7]. Once a proper hyper-parameter is chosen, the algorithm is very simple, takes up $\mathcal{O}(n)$ memory (where n is the number of parameters) and its per iteration costs are also very small. The main disadvantage of GD is that it is sensitive to poor scaling i.e. cases in which optimization problem's variables have vastly different influence on the function's output. Newton's method is unaffected by this as the Hessian matrix rescales the individual gradient elements so that each dimension θ_i of the parameter $\vec{\theta}$ is updated properly. This "scaling" is expressed as the condition number of

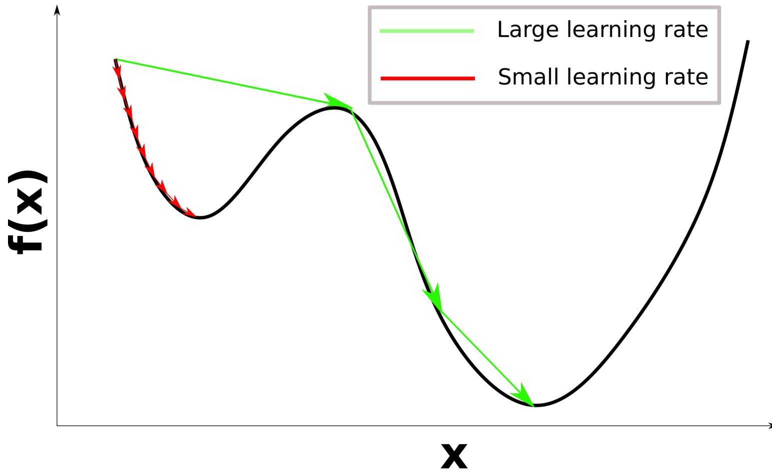


Figure 2.4: Schematic showing the effect of choosing learning rate for GD. Large learning rates result in more exploration but may also result in divergence. On the other hand, small learning rates result in exploitative behaviour of the nearby minima.

the Hessian matrix and is defined as $\left|\frac{\lambda_{max}}{\lambda_{min}}\right|$ where λ is an eigenvalue of the Hessian. The higher the condition number, the more difficult it becomes for GD to find the optimal solution [7].

In ML/DL problems however, there are a few caveats to using GD. Firstly, the objective function we are trying to minimize is a proxy for the actual function, which we do not have access to. Secondly, even if we consider the optimization problem in isolation, the datasets are so huge that computer memory limits them from being stored and used as a whole batch. Thus, GD is often impractical but for the simplest ML cases. To counteract this, a modified version called Stochastic Gradient Descent (SGD) is used widely, which calculates an estimate to the actual gradient by sampling from smaller batches from the dataset [7]. The randomness in the SGD optimizer is thought to perturb it and prevent it from settling to sub-optimal critical points [24]. But in the current framework, there is no dataset involved and thus, the gradient is not estimated from mini-batches but is calculated accurately using automatic-differentiation. This implies that the optimizer that was compared by Hoyer et al [3] with L-BFGS is in fact GD and not SGD and will be referred to as GD in this report.

2.3.2. Limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS)

One of the most popular second-order optimizers used in ML literature is the L-BFGS algorithm. This is a limited memory version of the quasi-Newton algorithm known as BFGS (named after its inventors). See Appendix A for details about the BFGS

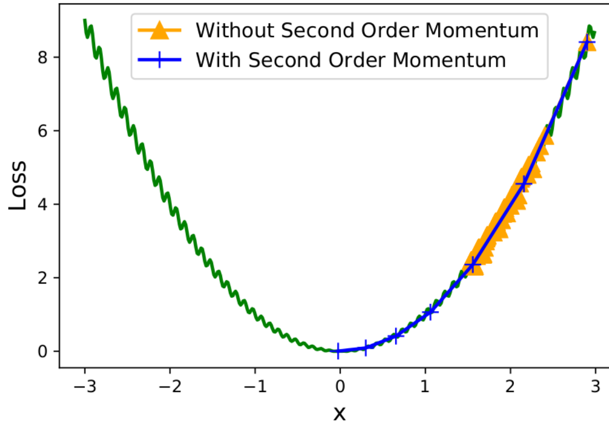


Figure 2.5: A noisy objective function obtained by adding sinusoidal noise to a quadratic function [26]. If the underlying global curvature information is utilized, an optimizer will be able to reach the global minima (Shown as the optimizer with second-order momentum). Other optimizers like BFGS can get stuck at one of the local minimas.

algorithm. Although the BFGS algorithm is highly successful in normal low-D non-linear optimization, its application to the high-D (high dimensional) cases found in DL is problematic for a few reasons. Firstly, since the algorithm needs access to all the vector pairs to form the quadratic model, the memory storage becomes an issue. This scales as $\mathcal{O}(n^2)$, where n is the number of parameters of the model. Secondly, the cost per iteration is $\mathcal{O}(n^2)$ arithmetic operations, which can be problematic for DL (in comparison to less costly first-order methods), when the dimension n can be millions or even billions. It is to rectify both these concerns that the limited memory version of BFGS, known as L-BFGS, came into being [25].

The L-BFGS algorithm differs from the BFGS algorithm in only a single aspect; the number of vector pairs stored, which now becomes a hyper-parameter. Intuitively, it makes sense that the gradient information from iterates further away from the current position should not be crucial. It may even be detrimental in certain cases as shown in Figure 2.5, where the global curvature information differs from the local one [26]. The overall algorithm is similar to algorithm 2, in fact the first "$m-1$" runs of both BFGS and L-BFGS are the same if they start with the same initial matrix, where m is the number of vector pairs to be stored. Since m is chosen to be a value between 3 and 20, the memory footprints are quite small even when n is large and scales as $\mathcal{O}(mn)$. Further, it reduces the per iteration costs to $\mathcal{O}(mn)$ as well [25]. These changes make it suitable for large-scale optimization.

The working of L-BFGS optimizer is described in algorithm 1. It can be roughly decomposed into 2 major components. The first is the computation of the step

Algorithm 1: L-BFGS algorithm

Given: initialization point $\mathbf{x}_0$;
k ← 0;
while *True* **do**
  Choose $\mathbf{M}_k^0$ as the initial approximation for this iteration;
  Compute search direction $\mathbf{p}_k = -\mathbf{M}_k \vec{\nabla} f_k$ using two-loop recursion [23];
  Use line search to find step size α ;
  Compute next iterate using Equation A.4;
  if $k > m$ **then**
    Discard oldest vector pair stored
  end
  Save the new vector pair $(\mathbf{s}_k, \mathbf{y}_k)$ to memory
  k ← k+1
end

direction, which involves scaling the gradient vector ($\vec{\nabla} f$) by the inverse of the approximation to the Hessian matrix i.e. $\mathbf{M} = \mathbf{B}^{-1}$, where $\mathbf{B}$ is the Hessian approximation maintained by the optimizer. At each iteration, this matrix is formed anew from an initial matrix using a recursion formula [23]. The initial matrix to start the recursion is usually chosen as a scaled-version of the identity matrix where the scaling factor depends on the vectors $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \vec{\nabla} f_{k+1} - \vec{\nabla} f_k$.

The second component is a line-search algorithm which decides the step size for the obtained step direction. In short, the first component tells the optimizer which direction to move and the second component tells it by how much. It does this by constructing a 1D approximation to the function along the step direction (using the available gradient and function information) and finding the minimum of this uni-variate function. An inexact line-search algorithm, which is used in L-BFGS, employs the strong Wolfe conditions ([23]) to determine the appropriate step size without finding the exact minimum of the 1D function. Because of these imposed conditions, the algorithm may require multiple calls to the objective function to choose the step size. In depth details for BFGS and L-BFGS are given in chapters 6 and 7 of [23].

2.3.3. Usage in DL Literature

In this section, a brief review of the usage of both GD and L-BFGS will be presented. The aim is to find out information about the kind of problems where each of them excel over the other. As mentioned in the preceding sections, the use of pure GD and L-BFGS in the DL literature is very sparse due to them being batch methods. So, this section will include their modified stochastic versions even though they may not be directly applicable to Hoyer et al [3]. For the sake of concreteness, the focus will be only on the optimization aspect and will thus include only the training runs for various ML/DL problems.

The usual applications of L-BFGS and its variants are usually in convex or strongly convex settings where there is access to high quality gradients (i.e. without too much noise). Noisy gradients can lead to poor curvature estimates which can have detrimental effects for the optimization [18]. Further, if the L-BFGS algorithm cannot accurately capture the Hessian matrix, either due to wildly varying curvature (as can be the case in some DL models [27]) or because of some particular structure of the Hessian which makes it less amenable to low rank approximations [28], L-BFGS is not expected to work well. SGD or its variants, on the other hand, thrives in ML/DL applications due to their simplicity, tunability, low computational costs and empirical success. Another success behind SGD is because of the role played by other strategies like pretraining, careful initialization schemes, learning rate scheduling which have made it quite robust and applicable to a general class of problems [26, 29]. Since the objective function landscapes are non-convex for most ML/DL problems, the convergence properties of second-order optimizers may not hold [28].

Bottou et al [18] provide an example where batch L-BFGS is contrasted with a well tuned SGD on a binary classification problem based on RCV1 dataset and a logistic loss function, as shown in Figure 2.6. The figure shows the characteristics of SGD; very fast initial convergence followed by stagnation. It is to be noted that this is solely because the use of mini-batches causes more parameter updates than the batch L-BFGS method which makes updates only once an epoch. The authors also note that, after a few epochs the training error achieved by L-BFGS would be lower than that of SGD. This is not an issue with ML cases where the focus is on the test error which may benefit from early stopping and hence SGD might be more beneficial.

One of the first papers to test both L-BFGS and SGD on DL problems was [28]. They tested L-BFGS using a mini batch with a forgetting procedure (reinitialization of L-BFGS after a certain number of iterations) and SGD with mini batches on training CNNs and autoencoders. They observed that L-BFGS performs much better than SGD on both models. The difference was more pronounced in the CNN case and authors remarked that this was due to the higher effectiveness of L-BFGS method when the number of parameters (≈ 10000 for the CNN case) are lower. For the autoencoder case, a better baseline was provided by SGD with a line-search since L-BFGS inherently uses line-search and it can be seen that both methods performed almost equally well, as shown in Figure 2.7. Thus, it may be that for non-convex settings, the line-search procedure may be of more importance than the optimizer itself. Also, note that although the test function curves are shown, it was mentioned in the paper that the training curves were similar. The relevance of these results to our case is to be evaluated keeping in mind the fact that the forgetting procedure for L-BFGS was used. This is a procedure followed to make L-BFGS better for non-convex settings and is thus not an off-the-shelf use of L-BFGS [30].

Li and Malik [31] tested L-BFGS and GD as baselines for their ”learning to optimize” approach. They tested both the optimizers on common ML tasks of logistic and linear regressions as well as on a small NN classifier. They found that for the convex logistic regression case, L-BFGS performed considerably better than GD and

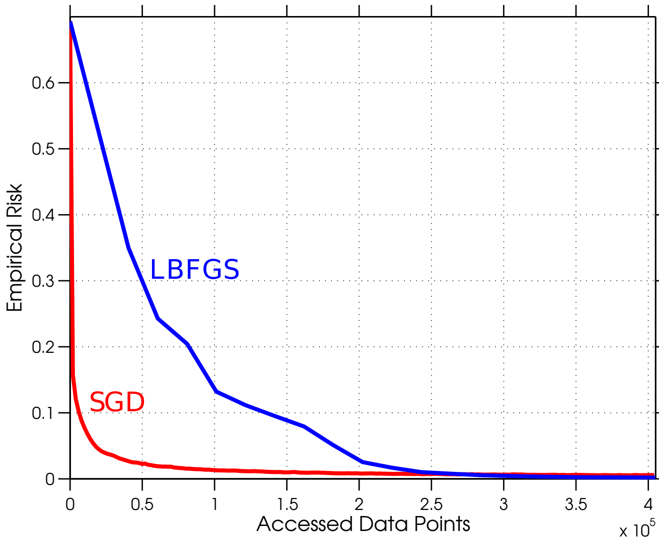


Figure 2.6: Comparison of minibatch SGD with L-BFGS for a binary classification task [18].

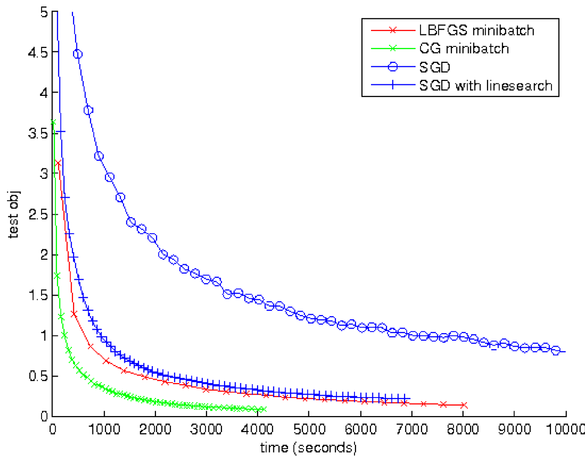


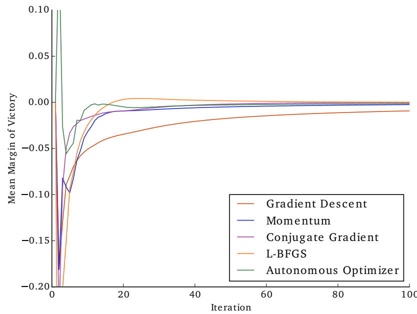
Figure 2.7: Test loss for autoencoder trained using minibatch L-BFGS with forgetting and minibatch SGD. SGD with a line-search provides a better baseline. The training loss behaviour was reported to be similar to test loss behaviour [28]. CG stands for the conjugate gradient optimizer.

converged in less iterations to better minima. The opposite happened in the case of robust linear regression which has a non-convex landscape, where L-BFGS was seen to diverge while GD performs well in comparison. For the case of the two-layer NN as well, which is known to have an even more complex landscape, L-BFGS was reported to diverge often. Figure 2.8 shows the conditions for the cases of non-convex NN classifier and convex logistic regression. The curves for linear regression, where the objective function is non-convex, are very similar to the NN case and is thus not shown. One interesting point to note is that, when L-BFGS does not diverge, the objective values obtained by L-BFGS is lower than that of GD.

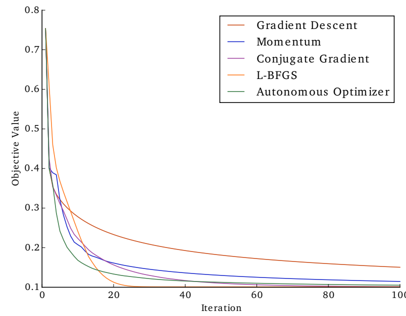
As opposed to the cases presented before, where the dimensions of the problem were comparatively lower than that of modern NNs (one of which is used in [3]), Dauphin et al [32] argues that in higher dimensional landscapes, most quasi-Newton optimizers (in their original form) would get stuck at high error saddle points and not local minima. The authors argued that L-BFGS, with its positive definite hessian approximation will not be able to escape such saddle points, where the approximation will no longer be valid. Further, they showed that for random high-D error surfaces, there was an exponential increase in the number of saddle points. Together, these would point that L-BFGS in its native form is not expected to provide the desired results, unless the landscape is somehow benign. In Hoyer et al [3] however, L-BFGS was reported to perform better on all problems than GD, without diverging for even a single tested seed. Perhaps, it can be hypothesized that the loss landscape of TO problems have some common characteristics that either promote some sort of convexity or some features that make it well-behaved for L-BFGS. It may also be that the number of high error saddle points are either quite low or inaccessible from common initialization schemes. These are some of the questions that need to be answered in this thesis.

2.4. Optimization landscapes

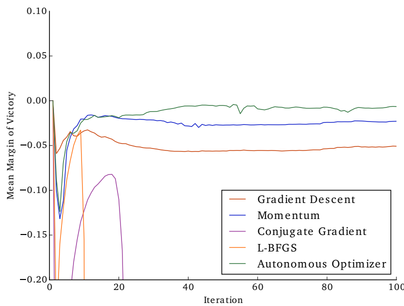
In studying the loss landscapes of DL models, there are broadly two streams, the first one that tries to provide answers through theoretical analysis and the second one through empirical testing on available datasets (MNIST [33], CIFAR10 [34] etc.) and State-Of-The-Art (SOTA) network architectures (ResNet [35], LeNeT [36] etc.). Due to the high dimensional (high-D) non-linear compound functions represented by NNs, a detailed theoretical analysis of the current SOTA networks and datasets is nonexistant in the literature. Instead, the authors of such papers focus on networks with one or two hidden layers or often make strong assumptions to make the analysis easier [37]. These assumptions may not be valid for practical settings [22]. So in this report, the empirical explorations will be the main focus. But general results obtained from theoretical studies will be used whenever necessary to support or discredit a particular paper's methodology. The empirical tools will roughly be demarcated as visualization tools and metric-based analysis tools. The first offers more qualitative information about the loss landscape while the latter is more quantitative.



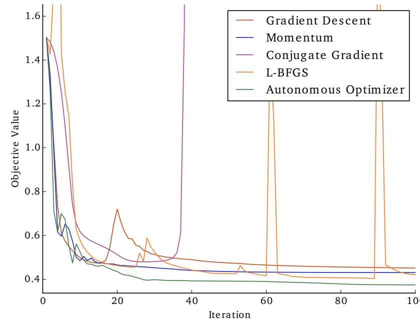
(a) Mean Margin of Victory for logistic regression.



(b) Training curves for logistic regression.



(c) Mean Margin of Victory for NN classifier.



(d) Training curves for NN classifier.

Figure 2.8: Comparing Batch L-BFGS and GD on convex and non-convex problems. Mean margin of victory is a metric that measures how low the final objective value is in comparison to other optimizers. The training curves are shown for one instance of the objective function [31].

2.4.1. Visualization tools

In the context of loss landscapes, an emerging trend is to study the geometric properties of the loss landscapes with the hope that geometry of the landscape and the interaction of the optimizer with this landscape will be crucial to understanding any optimization problem. Visualization tools are commonly employed because they convey qualitative information that is hard to gain from crunching numbers. But the caveat is that since the landscape formed by a typical Neural Network (NN) with millions or billions of parameters is extremely high-D, visualizing it is a challenge. Currently, researchers try to employ some kind of dimensionality reduction in order to visualize it either in 2D or 1D, while ensuring that these projections offer meaningful information about the high-D space. The idea is similar to constructing a 3D model from the drawings of its 2D projections, as is the case in the construction of buildings / machinery parts. But a word of caution is that extending the intuitions from low-D to high-D may not always work [19]. A note to consider before detailing the visualization techniques is that these are mostly tested for supervised learning settings and as such have not been tested for the settings in [3]. But by understanding the principles on which these methods are built, it is reasonable to extend them to the current settings.

1D projections or linear interpolations

One of the first and simplest forms of visualization was introduced by Goodfellow et al [38]. The method that they proposed was to train the network to find a minima (or any converged point) and then perform a linear interpolation between the initialization point and the obtained minima. Let $\vec{\theta}$ correspond to any of the points in the weight space where $\vec{\theta} \in \mathbb{R}^d$, where d is the number of parameters of the model. If $i = 0, 1, 2, ..., T$ be the number of iterations/updates of the weights, then $\vec{\theta}_i$ corresponds to the weight vector at a particular iteration i. In order to do linear interpolation between any two iterates, $\vec{\theta}_x$ and $\vec{\theta}_y$, the line joining the two iterates is parameterized by a scalar α such that:

$$\vec{\theta}_\alpha = \alpha\vec{\theta}_x + (1-\alpha)\vec{\theta}_y, \quad \alpha \in [0,1] \tag{2.7}$$

If $\mathcal{L}$ denotes the loss function chosen, then using suitable number of divisions in the interval [0,1], the loss value at each interpolated $\vec{\theta}_\alpha$ can be calculated. The set $(\vec{\theta}_\alpha, \mathcal{L}(\alpha))$ is plotted and this forms the linear interpolation. Although this is not the path taken by the optimizer, optimizers that employ line-search (e.g L-BFGS) "sees" a 1-D slice of the loss landscape at each iteration. The authors tested this visualization technique for most SOTA networks at that time on MNIST dataset using the optimized hyper-parameters of SGD. The interpolation between the initial and final points always resulted in a monotonically decreasing loss.

This 1D projection method can be extended to any two iterates as well. For example, Frankle et al [39] re-evaluated this method for modern networks using more complex datasets like CIFAR-10 and Imagenet by extending this method to not only interpolating between the initialization and final point but also between any of the iterates of SGD and the final point and found that even after a few iterations (a

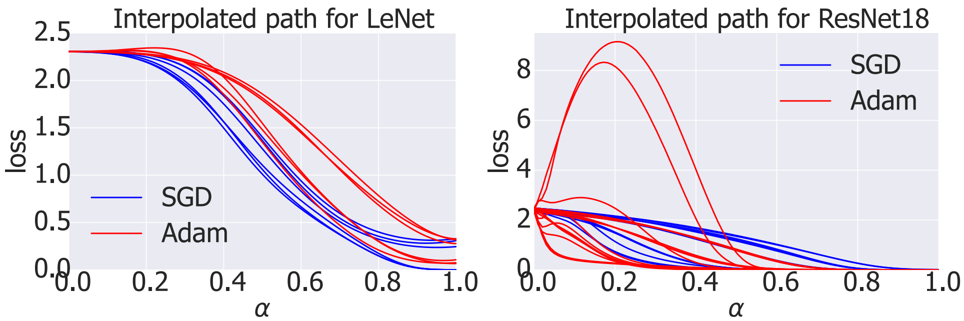


Figure 2.9: Linear interpolation made on CIFAR-10 dataset for two network architectures. The various lines correspond to different hyper-parameter settings of the optimizer that achieved at least 0.1 loss [41].

small percentage of the total number of iterations), there exits a a barrier when interpolating to the minima. Another use of this technique was demonstarted by Xing et al [40] to study the training dynamics of SGD on over-parameterized deep networks. They studied the loss landscape along the trajectory of the optimizer by interpolating between consecutive iterates i.e. if at a given iteration, the weights are $\vec{\theta}_t$ and after one update the weight vector becomes $\vec{\theta}_{t+1}$, then they use the formula in Equation 2.7 to interpolate between these points. This would provide a 1D slice of the loss landscape between the steps taken by the optimizer. They found that SGD stays above a "valley" with a certain height that is dependent on the learning rate chosen.

Although this method is simple and computationally inexpensive, Li et al [42] criticized this method by stating that it is difficult to visualize non-convexities using 1D plots and offered a better way to visualize landscapes using a technique called *filter-normalization* which is dealt in the next section. It is unclear why this comment was made because as seen in [41] and [39], these plots are indeed capable of showing non-convexities in the landscape. So, employing this technique creatively can help gather a lot of information, about both the optimizer and the landscape.

2D projections

While the 1D linear interpolation method gives a glimpse into the landscape along a 1D slice, 2D projections include another dimension and can help visualize planar slices of the high Dimensional space. Unlike the 1D case, this method provides more local information i.e. in the neighbourhood of the iterates. The modern version, that is widely used in literature, is the one proposed by Li et al [42]. They proposed that by using two random directions, normalized in a certain way, the actual loss landscape could be projected onto a plane such that it respects the symmetries of the network. They showed this empirically for RELU networks which have the property of invariance. If one layer weights are multiplied by a constant and the successive layer divided by the same constant, then the network is unchanged.

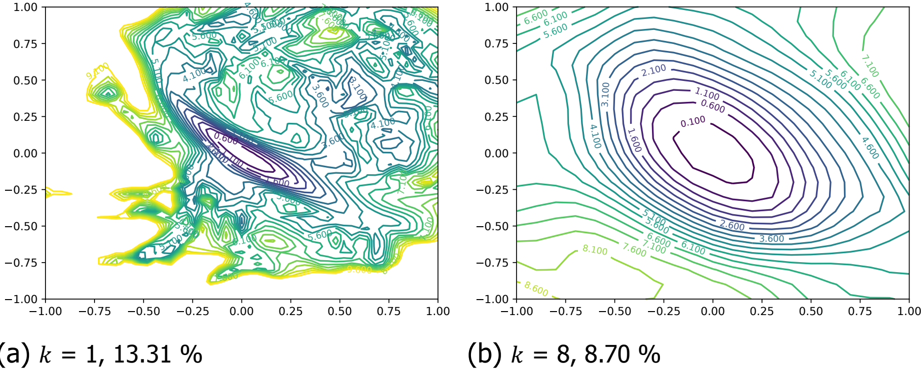
They demonstrated that such filter-normalized plots can capture this invariance while simple interpolations could not.

The proposed method works by first obtaining two random vectors with the same number of elements as the number of parameters of the network. A Neural Network's weights and biases have a particular structure that is dependent on the architecture. For example, a CNN's weights are the elements of the kernels (or filters) of a particular layer. When assimilating the entire weights, we obtain a list of matrices with a structure that corresponds to the filters in different layers. Filter-normalization procedure first involves converting the chosen random vector into the same structure as the list of matrices obtained. Let $\mathbf{D}$ be the list of matrices, with each entry populated from a Gaussian distribution and Θ be the list of matrices corresponding to the network. Then $\mathbf{D}$ is normalized such that each filter in $\mathbf{D}$ has the same norm as the corresponding filters in Θ. Mathematically, the replacement made is $\mathbf{D}_{i,j} = \frac{\mathbf{D}_{i,j}}{\|\mathbf{D}_{i,j}\|}\|\Theta_{i,j}\|$, where $\|.\|$ is the Frobenius norm , i corresponds to the layer and j to the filter in that layer.
Once the two random normalized directions are available, it is possible to plot the projection of the loss landscape along a plane spanned by these vectors at any point in the parameter space. Due to the high-D nature of NN landscapes, any vector chosen randomly is bound to be orthogonal to any other random vector with a very high probability, which makes it possible for them to uniquely define a plane. If $\vec{\theta}^*$ is the point of interest in the high-D space, then using this point as the centre of a plane with the flattened filter-normalized random vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ as the co-ordinate axes, we can choose a suitable grid and find the loss values at these grid points i.e. the plot is of the form $\mathcal{L}(\vec{\theta}^* + \alpha\mathbf{d}_1 + \beta\mathbf{d}_2)$, where α and β are the coordinates within the plane. Using this technique, the authors could identify a difference between two different network architectures based on the convex and non-convex regions found in the reduced plots as shown in Figure 2.10. They also proved empirically that if non-convexities are present in the reduced plot, they should also be present in the high-D space because the eigenvalues of the Hessian at any point in the reduced plot is the average of the eigenvalues in the high-D space. However, this reasoning cannot be extended to the convex regions, as even if the reduced plot shows a convex plot, the high-D space need not be convex.

Choosing the two directions that form the coordinate axes is very important. In [42], the authors chose *PCA* over random directions when plotting the optimizer trajectories, as the latter would not produce any noticeable trajectory. For finding the *PCA* directions, Li et al [42] considered each of the iterates excluding the minima ($\vec{\theta}_i \in \mathbb{R}^d$) as columns of a matrix. They then subtracted the final minima vector from these columns to center it at the minima and formed a data matrix. The PCA technique was then applied to this matrix and the first two principal components were used to form the 2D projection i.e.

$$\mathbf{d}_1, \mathbf{d}_2 = \text{PCA}([\vec{\theta}_0 - \vec{\theta}_T, \cdots, \vec{\theta}_{T-1} - \vec{\theta}_T]_{d\times T}) \tag{2.8}$$

where, $\vec{\theta}_T$ is the converged minima.



(a) k = 1, 13.31 %

(b) k = 8, 8.70 %

Figure 2.10: Filter-normalized 2D plots for Wide-ResNet-56 architecture without shortcut connections on CIFAR-10. k is a multiplication factor indicating the number of filters per layer and the number beside it is the test error [42]. a) It can be observed that for a narrow architecture, non-convexities are prevalent. b) When the network gets wider by a factor of 8, the non-convexities are more or less removed.

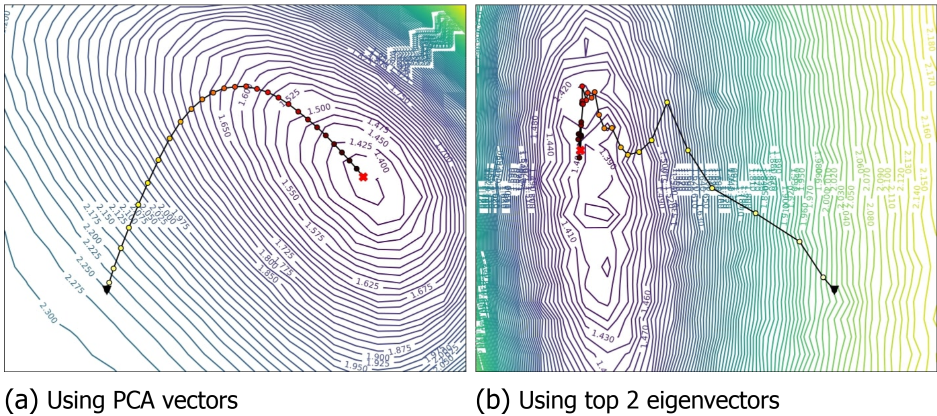
Later works by Chatzimichailidis et al [43] and Yao et al [44], considered using the top 2 eigenvectors of the Hessian instead. As for the eigenvectors, the authors utilized the Hessian of the loss with respect to the network parameters at the minima and found the two eigenvectors corresponding to the highest eigenvalues. Since, eigenvectors and PCA components are by definition orthogonal, these are plausible choices but the information gained from these plots have to be critically analyzed. For instance, Chatzimichailidis et al [43] points out that using PCA on SGD iterates always result in similar patterns. Thus, although visualization techniques based on dimensionality reduction can provide information about the real loss landscapes, their interpretation is not always clear unless backed up by some other metrics as well. An example of the plots made using these alternative coordinate choices is given in Figure 2.11.

2.4.2. Analysis tools

The empirical analysis tools in the literature are roughly of 2 types. The methods in the first category deals with the distances and various directions in the high-D space and their correlations. All of these methods utilize the iterates of the optimizer and is thus referred here as optimizer trajectory analysis. The other set of methods utilize the Hessian matrix and its properties to extract information about the optimization process. Both of these will be outlined in this section.

Optimizer trajectory analysis

A lot of papers investigating the loss landscape do it in conjunction with the optimization dynamics because they are both interlinked. Based on the metrics of the iterates of the optimizer, several tests and conclusions are made. Although these will deal only with the local landscape, when repeated for multiple random



(a) Using PCA vectors (b) Using top 2 eigenvectors

Figure 2.11: Alternative 2D projections. These plots were made by evaluating SGD optimizer on LeNet using CIFAR-10 dataset [43].

seeds, this can capture characteristic global information about the landscape and the optimizer as well.

Jastrzebski et al [45] mentions that, for SGD, the initial phase is important for the final performance and that the initial hyper-parameters of the optimizer determine the rest of the optimization trajectory and especially the conditioning of the Hessian along the trajectory. This could be the reason why various papers track various metrics of the optimization trajectory and try to find correlations between the final performance and these chosen metrics.

Among the metrics tracked are the gradient [45] and weight norms [42], cosines of angle between various directions or vectors in the loss landscape or the distances of vectors from one another. For example, in [40], the authors measured the angle between successive gradients at the iterates of SGD and GD for VGG networks on CIFAR-10. A value of -1 for GD was attributed to oscillation above a minima valley. The authors also tracked the distance from the initialization point to conclude that SGD explores more than GD. In [46], the authors chose to project the gradient at an iterate onto the direction of the converged solution. i.e. If $\vec{\theta}_j$ is the parameter vector of the model at a given iterate and $\vec{\theta}_T$ is the converged solution, the metric is $\vec{\nabla} f_j \cdot (\vec{\theta}_T - \vec{\theta}_j)$. Using this information, they showed that in networks without batch normalization, gradient was uninformative and this metric was very close to zero. Many more similar metrics can be contrived with respect to various important directions like the step direction, gradient direction, direction to the minima, eigenvectors of the Hessian, PCA directions etc. which may provide more information about the optimization process. For example, once the eigenvectors of the Hessian are found, their relationship with the gradient vector can be studied, as done in [47]. They also studied the amount of overlap of the gradient with the Hessian's

top subspace by measuring a proxy metric given by

$$\text{Overlap} = \frac{\mathbf{g}^T \mathbf{H}\mathbf{g}}{|\mathbf{g}| . |\mathbf{H}\mathbf{g}|} \tag{2.9}$$

2

where $\mathbf{g}$ is the gradient vector and $\mathbf{H}$ is the Hessian matrix.

Hessian based metrics

Analysis of the Hessian matrix of deep NNs for information regarding the loss landscapes is an emerging trend. Traditionally, this task was not amenable as storage of the Hessian matrix for common architectures was impossible as the storage was $\mathcal{O}(n^2)$, where n is the number of parameters of the network. But in recent papers, ideas from Numerical Linear Algebra (NLA) and Randomized NLA which make the Hessian analysis computationally feasible have been imported to ML/DL settings [44]. The main components of all the papers that utilize the Hessian of the loss w.r.t the NN's parameters are:

1. Hessian-vector products (Hvps)
2. The Lanczos Algorithm

Both of these will only be briefly explained and the reader is referred to [44], [43] or [46] for an exact mathematical treatment.

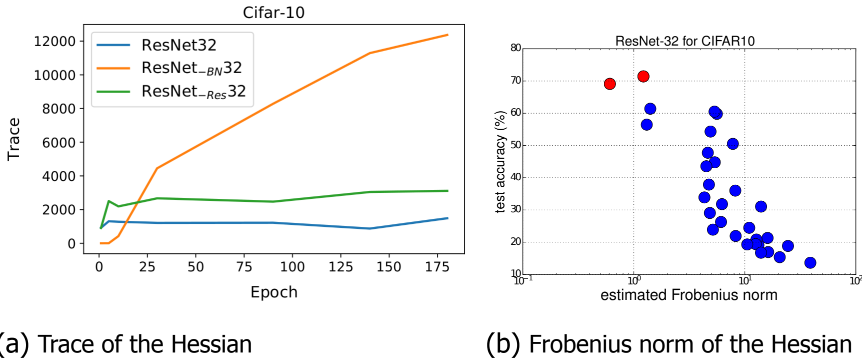
For easy access to Hvps without explicitly forming the Hessian matrix, Pearlmutter's *R* operator [48] can be used which can be easily implemented once the computational graph for the gradient calculation is defined and requires $\mathcal{O}(n)$ storage only. With access to this oracle, many properties of the Hessian can be easily extracted. For example, the Hutchinson algorithm [44] can be used to find the trace of the Hessian as an expectation value. Mathematically, for random vectors $\mathbf{v}$ from the Rademacher distribution or standard Gaussian distribution, we have

$$Tr(\mathbf{H}) = \mathrm{E}_{\mathbf{v}}[\mathbf{v}^T \mathbf{H}\mathbf{v}] \tag{2.10}$$

When tested on ResNets trained on CIFAR-10 with SGD with momentum, it was shown that for shallower networks, the trace increases throughout the training process while for deeper architectures with Batch Normalization, trace decreases as training proceeds. This indicates that the trace of the Hessian is a important metric that can be tracked throughout training to gain valuable information regarding the landscape. Similarly, Wu et al [49] mentions a formula for computing the Frobenius norm of the Hessian as an expectation using

$$||\mathbf{H}||^2 = \mathrm{E}_{\mathbf{v}}[|\mathbf{H}\mathbf{v}|^2] \tag{2.11}$$

and uses it to estimate the volume of attraction of a minima and concluded empirically that the volume of attraction of bad minima are exponentially smaller than that of good minima for common networks and datasets. Figure 2.12 shows the metrics of trace and norm from the corresponding papers. With the Hvp oracle, one can



(a) Trace of the Hessian (b) Frobenius norm of the Hessian

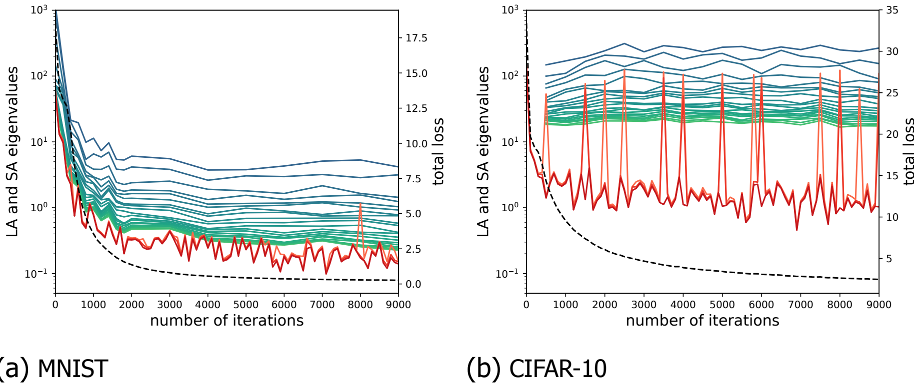
Figure 2.12: Estimated hessian metrics tested on ResNet-32 architecture on CIFAR-10 dataset using SGD a) Trace is affected by the architectural changes [44]. b) Norm at the minima is correlated with the test accuracy. The red bubbles represent the case when the dataset was not poisoned (not relevant to the discussion) [49].

also extract either the top or bottom k eigenvalues and eigenvectors where $k << n$. These eigenvalues were tracked in [50] where they showed that the top eigenvalues decrease and stabilize early while the most negative eigenvalue did not. The authors also tracked the curvature along the eigenvector at a particular iterate to check the stability of the eigen space i.e. If the eigenvector at iterate $\mathbf{x}_j$ is $\mathbf{e}_j$, then the curvature along this direction is plotted using the Hessians at all other steps. The chosen eigenvector is randomly chosen from the set of iterates. The curvature along a direction $\mathbf{d}$ is given by:

$$\text{Curv}(\mathbf{d}) = \mathbf{d}^T(\mathbf{Hd}) \tag{2.12}$$

They found that the top eigenspace is stable while the negative eigenspace is not and is dependent on the dataset, ashown in Figure 2.13. Their argument was that second-order methods utilize the Hessian at the previous iterate to pre-condition the gradients at the current step. If the Hessian is stable, this can lead to faster convergence for second-order methods. But since L-BFGS maintains only a positive definite representation of the Hessian, the exact consequences of the instability of the negative eigenspace is unknown. Further, Fort et al [51] proposed that the ratio $r = \text{Tr}(\mathbf{H})/\text{Norm}(\mathbf{H})$ is an indicator of the suitability of initialization. Since the trace and norm of the Hessian measure the average curvature and the deviation in the curvature along random directions at a given point in the loss landscape, the authors argue that if this ratio is much greater than 1, then this is a region of extremely high positive curvature and is suitable for faster training. They termed the spherical zone in the loss landscape where this was valid as the Goldilocks Zone and showed that when training on random hyperplanes or hyperspheres, the overlap with this region promoted their success.

The second component, i.e. Lanczos algorithm, can be used to extract not just a few eigenvalues but to form an approximation to the exact eigenvalue density spectrum



(a) MNIST

(b) CIFAR-10

Figure 2.13: Estimated LA (Largest Algebraic, in blue/green) and SA (Smallest Algebraic, in red) eigenvalues of the Hessian. The dashed line is the loss curve [50]

(EDS) of the Hessian with high accuracy and computational feasibility. The method for forming the EDS is through a series of approximations and is collectively called the Stochastic Lanczos Quadrature (SLQ) [43], [44]. The first idea behind SLQ is to approximate the Hessian EDS which is a discrete delta function by convolving a gaussian kernel f with standard deviation σ i.e.

$$\phi(t) = \frac{1}{n}\sum_{i=1}^{n}\delta(t-\lambda_i) \approx \frac{1}{n}\sum_{i=1}^{n} f(\lambda;t,\sigma) = \phi_\sigma(t) \tag{2.13}$$

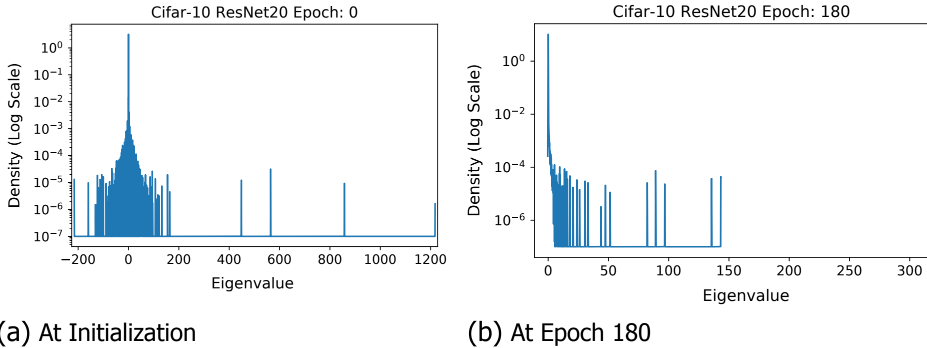
where λ are the eigenvalues of the Hessian matrix. Now that a continuous function has been formed, this is evaluated using a type of quadrature rule involving o nodes and weights. The nodes and weights are found using the Lanczos algorithm starting from a random vector and using the Hvps. In essence, the problem of finding the EDS of the $n \times n$ hessian is converted to the simpler problem of finding the EDS of the much smaller $o \times o$ approximation. For the Lanczos algorithm, the estimate for a single run is given by

$$\phi_\sigma^v(t) = \sum_{i=1}^{o} w_i f(\tilde{\lambda}_i;t,\sigma) \tag{2.14}$$

where the weights and node points are w and $\tilde{\lambda}$ respectively. This is repeated for k different seed random vectors and in expectation, we get

$$\phi(t) \approx \phi_\sigma(t) \approx \frac{1}{k}\sum l = 1^k \phi_\sigma^v(t) \tag{2.15}$$

For SLQ, the hyper-parameters are o, σ and k. The authors in [46] showed mathematically and empirically that the approximations are valid and the error decreases exponentially as o in increased for a fixed value of σ and k. Also, the approximations are better when the dimension n of the problem is higher as well, which



(a) At Initialization

(b) At Epoch 180

Figure 2.14: Eigenvale Density Spectrum (EDS) of the Hessian matrix of ResNet-20 trained on CIFAR-10 dataset. The x axis represents the eigenvalues of the Hessian while the y axis represents their normalized density [44]

makes it suitable for large networks. For typical NNs, the authors used k = 10, o = 90 and σ^2 = 1e-5 and stated that this was an extremely conservative choice and resulted in double precision accuracy. For easier computation, o, k and σ can be chosen to be lower values without loosing much accuracy. With the same setup, information from [52] can be used to determine the trace of the resulting matrix after any function acts on it i.e. $Tr(f(\mathbf{H}))$ can be easily calculated.

The EDS of the Hessian was studied for smaller networks by Sagun et al [53] who showed that at the minima, the Hessian contains a bulk of eigenvalues concentrated near zero and a few large magnitude outliers. [43] calculated the EDS at each step of the optimization trajectory and found similar conclusions on LeNeT trained on CIFAR-10 with SGD momentum. They also noted that at initialization, the spectra is almost symmetrical and that the negative eigenvalues decrease rapidly but are never exactly zero, indicating that the final point is not a strict minima but more likely to be a saddle point. This evolution of the EDS is shown in Figure 2.14. Ghorbani et al [46] experimented with ResNets and VGG networks and came to the conclusion that for SGD the outlier eigenvalues slow down the optimization process by concentrating the gradient energy in the high curvature subspace and not fully exploring the flat regions of the loss landscape. They also found that architectural changes such as the addition of Batch normalization has an effect of the spectrum by which the larger eigenvalues are pulled closer to the bulk.

3

Understanding neural optimization of topology

ABSTRACT [1]
Inverse design with topology optimization has followed the same computational graph for decades. The unknown material density is distributed within a domain, a computational analysis predicts the response of that design and its derivative with respect to the unknown, and this information is used by a chosen gradient-based optimization algorithm to find the next design iteration until it reaches an optimum (local or global). Recently, however, a counter-intuitive strategy was proposed which augments the computational graph by including a neural network in between the response prediction (computational analysis) and the generation of a new design (image). This shifts the optimization problem from its original space to the weight space of the neural network. Yet, this indirect optimization process was shown elsewhere to outperform conventional topology optimization for a large number of structural compliance problems – at least when choosing a particular convolutional neural network (part of a U-Net) and a particular optimizer (L-BFGS). This investigation provides quantitative and qualitative arguments that justify why these choices are successful, concluding that the line-search component of L-BFGS is key to traversing the reparameterized objective (loss) landscape and quickly reaching good solutions in "flat" regions of the landscape. Importantly, these topology optimization problems are not stochastic which make them different from the majority of conventional deep learning applications, favoring the use of line-search. Similarly, although to lesser extent, the approximation of the Hessian provided by L-BFGS helps moving more effectively within the flat regions by rescaling the gradients, the quality of the approximation is less relevant. Together with the deep image prior effect associated to deep learning, these arguments explain the early success of the neural reparameterization strategy in topology optimization, in spite of the non-convex objective landscape distortion that they introduce even for landscapes that were originally convex.

[1]This chapter is formatted for submission as a journal article authored by Suryanarayanan M.S., G. Kus and M.A. Bessa.

Neural reparameterization of topology optimization (TO) problems was introduced by Hoyer et al [3] and it has been investigated by others, e.g. [4]. This method involves shifting the optimization process in TO from the physical system of parameters to the weights and biases of a deep neural network (DNN). Conventionally, in the density based formulations of TO using the Solid Isotropic Material with Penalization (SIMP) approach, a well-known gradient-based optimization algorithm progressively optimizes a structure that is parameterized by density values at discrete regions (usually finite elements) [15]. By performing neural reparameterization on TO problems with a compliance objective, the referred authors [3] showed that conventional TO could be outperformed for the majority of problems tested by achieving lower compliance values in fewer iterations. This is discussed in detail in subsection 2.1.1.

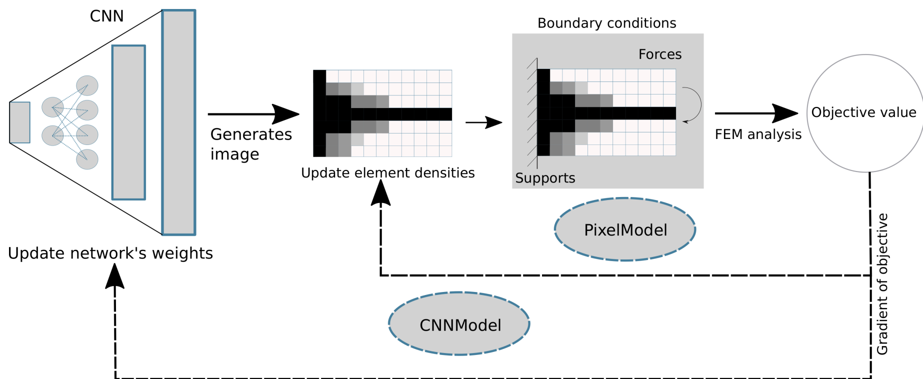


Figure 3.1: A simplified schematic of the neural reparameterization method applied to compliance TO, as adopted in [3]. The PixelModel is the conventional way of solving the TO problem (in the pixel space) while the CNNModel is the proposed method (in the neural space).

Nevertheless, effectiveness of the neural reparameterization strategy in TO depends on previously known factors such as problem size, dimensionality and target quantity of interest, but also on new ones that are introduced by the neural reparameterization trick, namely the choice of neural network architecture and its hyperparameters, as well as the optimizer for determining its weights and biases. All these parameters and hyperparameters define an objective (or loss) landscape of the TO problem which then needs to be traversed in the hope of reaching its optimum value as quickly as possible. This work focuses on understanding why the best performing optimizers in the context of neural reparameterization are different from the ones typically used in deep learning problems. This is achieved by leveraging novel objective landscape visualization and metric computation techniques developed in the deep learning literature (see chapter 2 for details on these techniques).

A short remark in the original article [3] mentioned that the L-BFGS optimizer [54] seemed to be more suitable for the neural reparameterization strategy of TO when

compared to Gradient Descent[2] (GD) [23]. On first glance, this observation is surprising because GD algorithms (including SGD) are the *de facto* standard to solve most deep learning problems [7]. Since no quantitative proof or further elaboration on the topic was provided, this investigation is dedicated to finding an explanation for this observation to reveal new routes of improvement for this nascent field.

Any successful optimization process requires the synergy between the optimizer and the objective landscape on which it travels. The overall optimization process is better understood by comparing and contrasting the interaction of different optimizers with the same underlying neural landscape. The L-BFGS and GD algorithms as well as the tools to analyze the objective landscape are reviewed in chapter 2. The methodology followed throughout is to modify the algorithms and isolate parts of the optimizers to determine the crucial components that explain their success, including gradient approximation, Hessian approximation and line-search capabilities. In addition, tools and techniques to measure the optimizers' interaction with the landscape are used to provide quantitative and qualitative arguments that explain their advantages and limitations. Landscape assessment is a quickly developing field in deep learning literature, see e.g. [20]. Finally, the differences or similarities shared by these interactions on diverse test cases is investigated.

Figure 3.2 summarizes the 16 test case combinations considered herein. *CNNModel* refers to whether the problem is solved with the neural reparameterization trick using a convolutional neural network (CNN), while *PixelModel* refers to a conventional SIMP TO problem. Four practical TO problems with specific boundary conditions and constraints are tested, as named in the figure. For each problem solved by each TO strategy, the penalization factor p of SIMP is assumed to be 1 or 3 because this transforms a convex ($p = 1$) problem into a non-convex ($p = 3$) one when considering the conventional TO strategy.

The selection of the four TO problem instances was made randomly, but choosing two different finite element mesh sizes (pixel number) – see also Figure 3.1. Since the neural architecture that generates the image of the structure is inherently tied to the size of the problem, the problems sharing the same size have the same architecture (same number of output neurons). It was decided to choose 2 different sizes (one smaller and the other larger) because larger problems were reported [3] to benefit more from the reparameterization scheme. The forward pass through the finite element analysis (the physics part of the computational graph) is expensive and to limit the number of test cases, 32x64 elements were chosen for *michell centered beam* and *pure bending moment* while 128x128 elements were assigned for *Crane* and *Causeway bridge*. The four TO problems are given codes in Table 3.1 to facilitate their reference.

Each solution obtained for each strategy for a given problem and penalty factor

[2]The authors actually mentioned that L-BFGS seemed better than Stochastic Gradient Descent (SGD), but since there is not stochasticity in the tested TO problems, the work presented herein only refers to Gradient Descent (GD).

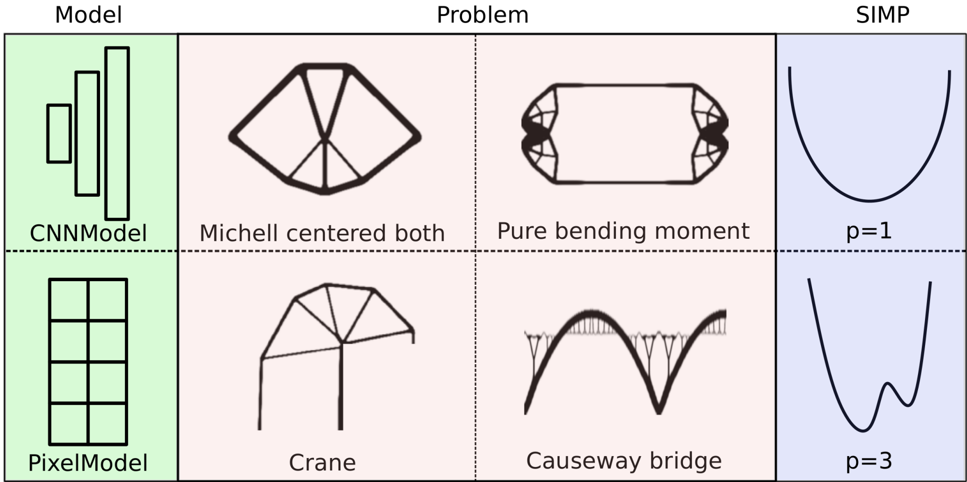


Figure 3.2: The 16 test case combinations comprising of 2 models, 4 problem formulations and 2 values of penalization *p*. The top two problems are small with 32x64 elements (small) while the bottom two are large with 128x128 elements.

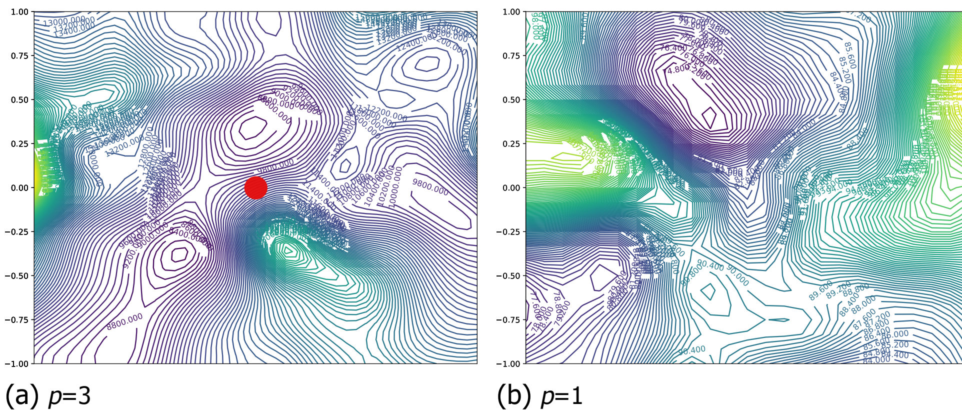
is the median over 10 random initializations. This mitigates the dependence on problem parameter initialization. Scipy's [55] implementation of L-BFGS is used herein, since it provides a wrapper to the original Fortran code of L-BFGS-B [54]. Although L-BFGS and L-BFGS-B are conceptually the same, their implementation is different from each other, as discussed in section B.4. The two terms are used interchangeably for the purposes of this work. Concerning GD, the implementation of Keras [56] is followed. The ideal learning rate was chosen for each problem by performing a coarse grid search for hyper-parameter optimization. Details about the optimizers and the hyper-parameter optimization performed for GD are available in section B.7.

Problem code	Problem name	Size
A	Michell centered beam	32x64
B	Pure bending moment	32x64
D	Causeway bridge	128x128
E	Crane	128x128

Table 3.1: The tested problems and their codes. These codes will be used to make references to the problems. More details about the boundary conditions for these problems is given in subsection B.1.3

Figure 3.4 shows the median of the results obtained for large problems (i.e. problems D and E in Table 3.1), while Figure C.2 shows the same for small problems. Focusing on the top row of Figure 3.4, i.e. when considering a convex ($p = 1$)

objective landscape for the *PixelModel*, one observes that all initializations lead to the same loss value for both GD and L-BFGS, as seen in Figure C.3 for the *CNNModel* and a similar figure exists for the *PixelModel* (not shown for brevity, see Figure C.4). For the PixelModel, this is expected due to the convexity of the landscape (excluding the volume constraint imposition), but it could be different for the CNNModel because the landscape is expected to be non-convex. This non-convexity is confirmed when visualizing the objective landscape of the CNNModel via the filter-normalization technique of Li et al [42], clearly exposing the high dimensional landscape surrounding the initialization point. In brief, this method defines a plane using two normalized random directions and projects the landscape onto this plane (see Figure 2.4.1 for more details). A plot that is representative of both $p = 1$ and $p = 3$ test cases for the CNNModel is shown in Figure 3.3, showing the clear non-convexities surrounding the initialization.



(a) p=3 (b) p=1

Figure 3.3: 2D projection of the loss landscape surrounding the initialization of the optimizer. The landscape has been projected onto 2 filter normalized random vectors [42]. The red dot at the centre is the starting point of the optimizer trajectory (not shown for Figure 3.3b). This figure was obtained by testing problem D on CNNModel using L-BFGS.

The convexity argument helps explaining why L-BFGS is superior to GD for the *PixelModel* because L-BFGS is known to be faster for convex problems (see subsection 2.3.3). However, the results for the *CNNModel* are less obvious, since in the neural space both GD and L-BFGS perform similarly on the problems with $p = 1$ both in terms of number of steps to convergence and obtained solution (lowest compliance achieved). Interestingly, for these problems with $p = 1$ the number of iterations to reach the converged solution is lower for the *CNNModel* than the *PixelModel*, despite the non-convexity of the *CNNModel* overall landscape.

When considering the case of $p = 3$ for the CNNModel, one verifies the claim that L-BFGS performs better than GD. This remains true even when conducting the optimization with GD for more than 2000 steps, i.e. far more than what is shown in the bottom part of Figure 3.4 (and Figure C.2). The median of the final

3

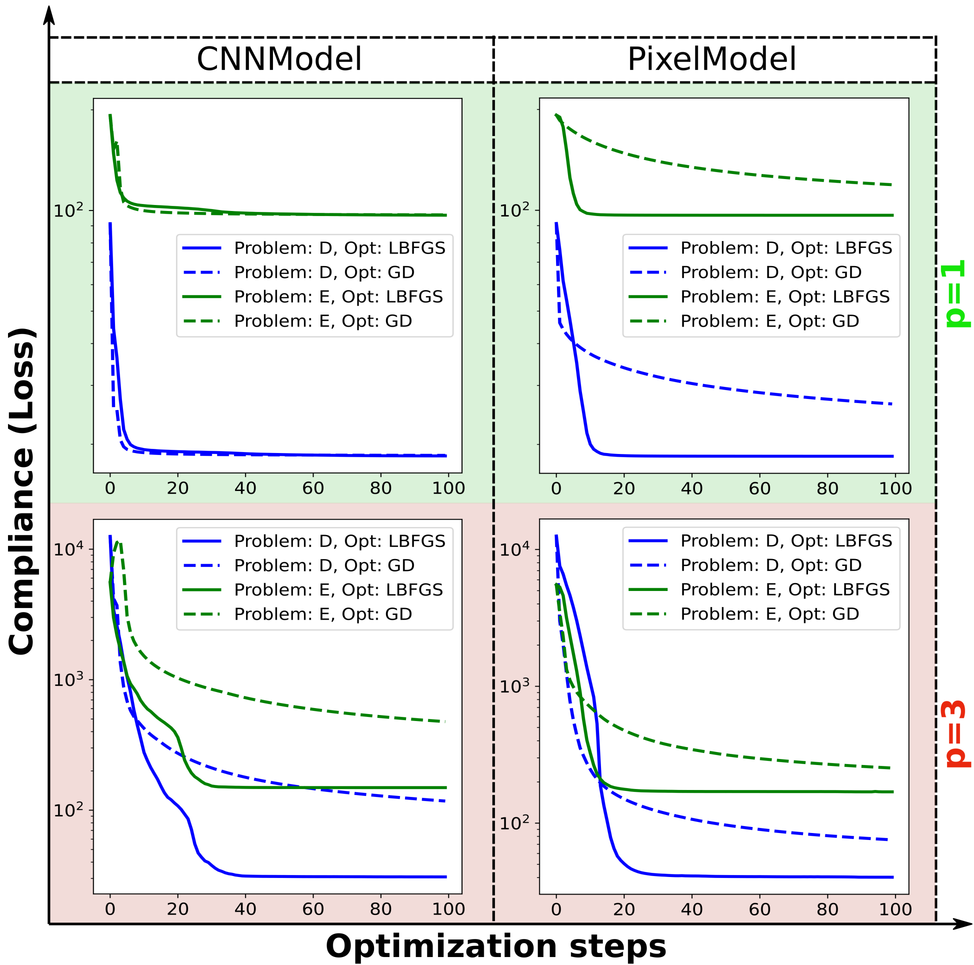


Figure 3.4: Loss curves comparing PixelModel and CNNModel when the larger problems D and E were optimized using L-BFGS and GD. The top row corresponds to the case with penalization = 1 and the bottom row is for p=3. The dashed lines represent GD while solid lines are for L-BFGS.

loss values for GD are shown in Table 3.2. Also, even with the best step size selection, the solution obtained with GD diverges for certain randomization seeds when considering the *CNNModel* but not for the *PixelModel* (see Figure C.1). Clearly, in both models with a non-convex objective, L-BFGS is superior to GD even though the difference is less pronounced for the PixelModel. Noteworthy to mention that the solution is reached in slightly fewer steps for the *PixelModel* than the *CNNModel*, but the converged value is lower for the *CNNModel*.

These results confirm that although the neural landscape is non-convex, the con-

Problem	Optimizer	CNNModel	PixelModel
A	L-BFGS (300 steps)	54.61	76.12
	GD (700 steps)	79.67	89.49
	GD (2000 steps)	73.9	78.04
B	L-BFGS (300 steps)	8.54	9.16
	GD (700 steps)	13.62	9.88
	GD (2000 steps)	11.74	9.28
D	L-BFGS (300 steps)	30.68	39.72
	GD (700 steps)	61.36	46.53
	GD (2000 steps)	49.34	40.34
E	L-BFGS (300 steps)	148.47	169.07
	GD (700 steps)	275.93	175.36
	GD (2000 steps)	231.65	163.01

Table 3.2: Comparison of the median of the loss values when GD is run for 700 steps and 2000 steps for penalization = 3 case. The data for L-BFGS is also given for reference. The best value obtained across all the presented combination is underlined.

vexity of the conventional TO problem objective ($p = 1$ or $p = 3$) plays a major role in determining the performances of different optimizers on these landscapes. The success of many deep learning applications where the loss function is convex (e.g. MSE loss, cross entropy loss [7]) is possibly tied to these observations. An interesting observation is that navigating these neural landscapes seems to be definitely easier for L-BFGS without much regard for the convexity of the objective. Finally, these results suggest that neural reparameterization is favourable to the TO process by allowing L-BFGS to find better solutions (lower compliance) than in the pixel space, despite the distortion to a non-convex objective landscape.

A possible explanation for these results is that neural reparameterization, by virtue of its over-parameterization, creates a landscape with a proliferation of low loss "minima" (in fact, these are low loss basins, as will be discussed later). This can offer a suitable optimizer an easier access to one of these minima. To qualitatively confirm this, the linear interpolation techinque (see subsection 2.4.1) was used for the *CNNModel* between various points in the high dimensional space. This method, which creates a 1D slice of the landscape, reveals a "seemingly convex" and smooth subspace for most of the test cases, as illustrated in Figure 3.5, giving credence to this explanation.

This idea is also supported from recent findings in the deep learning literature. Firstly, Sagun et al [53] reported that high loss minima traps do not occur in such over-parameterized regimes. Secondly, Garipov et al [57] and Draxler et al [58] observed that the minima in neural nets are not isolated but are connected by low-

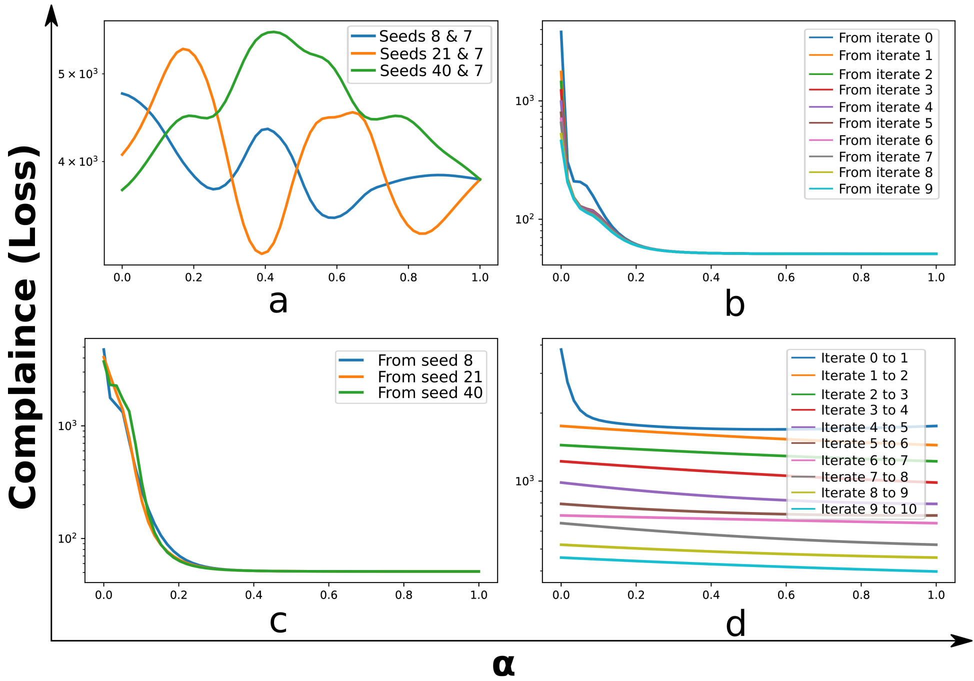


Figure 3.5: Various plots using the linear interpolation tool [38]. These plot represent the linear interpolation a) between two random initialization points showing a highly non-convex function b) between iterates of the optimizer to the obtained minima (minima at $\alpha = 1$) c) from random initialization points to the obtained minima (minima at $\alpha = 1$) d) between successive iterates. This plot was obtained for problem A, CNNModel, $p = 3$ and L-BFGS optimizer.

loss paths. Finally, the concept of *intrinsic dimension* of an optimization problem, as coined by Li et al [19] and explained in section A.2 is also relevant to this discussion. Reparameterized compliance TO problems which seem to be "easier" to optimize might have a low intrinsic dimension. Because of the large over-parameterization, the solution-set, which would be a low dimensional subspace having this intrinsic dimension, will have a very high redundancy. These are plausible expanations that justify the higher performance of the *CNNModel* when compared to the *PixelModel* even for convex objective landscapes prior to the reparameterization. However, these arguments do not answer why L-BFGS is more successful than GD in the particular setting of neural reparameterization.

Recall that L-BFGS differs from GD in two aspects. First, when choosing the step direction, L-BFGS forms an approximation (**B**) to the Hessian matrix (**H**) to precondition the gradient. Second, for choosing the step size, L-BFGS uses a line-search algorithm (see subsection 2.3.2). Since the curvature information encoded in the Hessian is what enables faster convergence, comparing the actual Hessian with

the approximation made by L-BFGS is expected to provide information about which component is more important.

The approximate Hessian constructed by L-BFGS can be compared with the exact Hessian using Stochastic Lanczos Quadrature [46]. This method allows the construction of the approximate eigenspectrum of a matrix and is especially well suited for large matrices (such as those occurring in DNNs) due to its computational and storage efficiency. For this purpose, the code from [46] was modified to fit the neural reparameterization framework (see section B.5 for more details). The oracles for implicitly calculating the matrix-vector products were implemented. For the Hessian, this required a twice differentiable computational graph and Pearlmutter's trick [48]. The **B**-vector product was derived from the algorithm of L-BFGS, as mentioned in [54]. The details for these are given in section B.4.

This method was employed to compute the Eigenvalue Density Spectrum (EDS) for each of the test cases but the complexity of the spectra made this comparison difficult. The reason for this is because L-BFGS's hyper-parameter "*m*", which is the number of BFGS updates made to approximate the Hessian, was set to a value of 10 and would result in a complicated EDS. In order to simplify the experiment, it was decided to witness the effect of having a Hessian approximation obtained using fewer BFGS updates. This is achieved by changing the "m" hyper-parameter from the default case of $m = 10$ to $m = 1$, keeping in mind that the recommended values in the literature is between 3 and 20 [23]. Choosing the hyper-parameter "*m*" as zero would not perform the Hessian approximation at all and so, m=1 was chosen as the candidate. It can be argued that the algorithm should become less robust and is not expected to perform as well as the default case since a poor approximation of the Hessian will be made.

To our surprise, the algorithm was robust to this change, as shown in Figure 3.6 and Figure C.5. It can be clearly observed that for the CNNModel, for all the cases, the loss values are exactly the same as that of L-BFGS. This could be either due to the fact that the Hessian of such reparameterized TO problems is easily estimated or it can also be that an accurate approximation is not a necessity to optimize these problems. The same observation holds for almost every case of the PixelModel, except for problem D with p=3 where there is a small discrepancy (See Table 3.3). As a result of this lack of sensitivity to L-BFGS's approximation of the Hessian the representative EDS spectra shown in Figure 3.7 pertains the algorithm with $m = 1$ (referred to *LBFGS_M1*). Similar plots were obtained for all cases considered, as shown in Appendix C. Since the optimizer manages to perform well despite a poor approximation of the actual Hessian at the various steps of the optimization process, these results provide a hint that the other component of the L-BFGS optimizer (line-search) is more important.

To test the relative importance of line-search and the Hessian approximation capability, the two components are isolated to derive two simplified versions of the L-BFGS optimizer. The first one does not have the Hessian approximation capability, i.e. the optimizer simply performs *gradient descent with line-search* and is referred

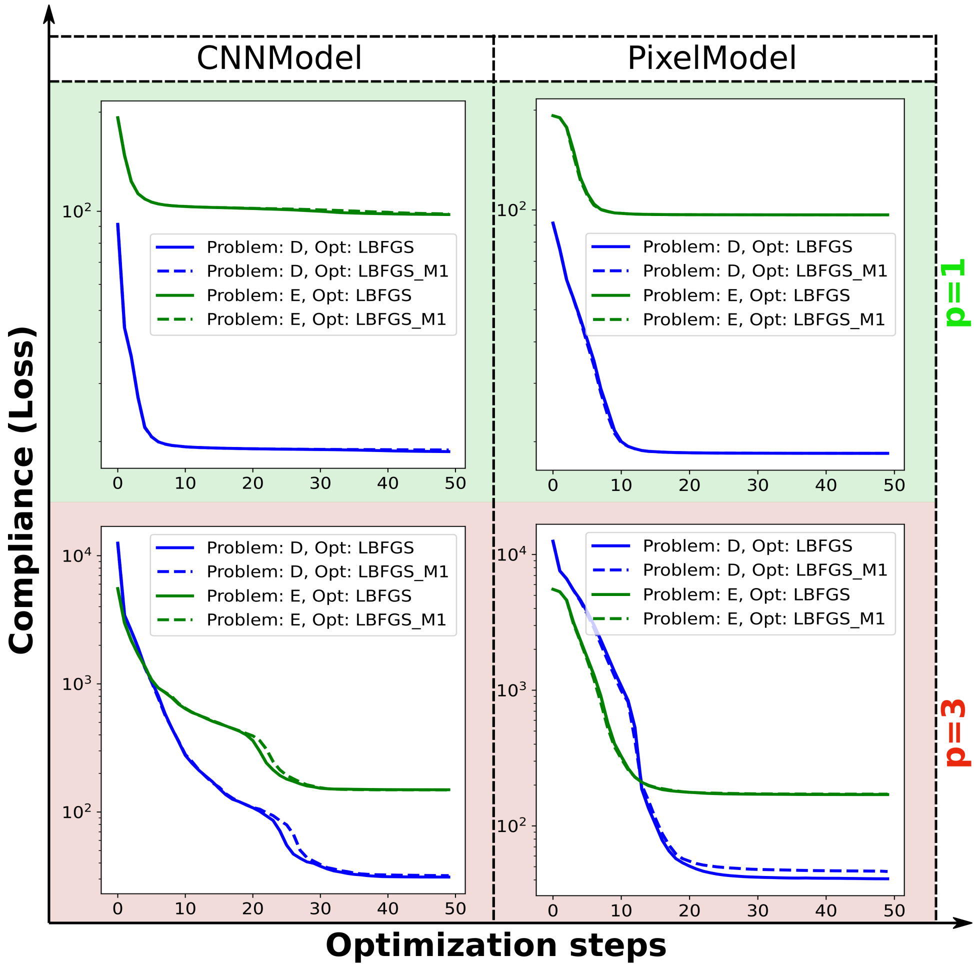
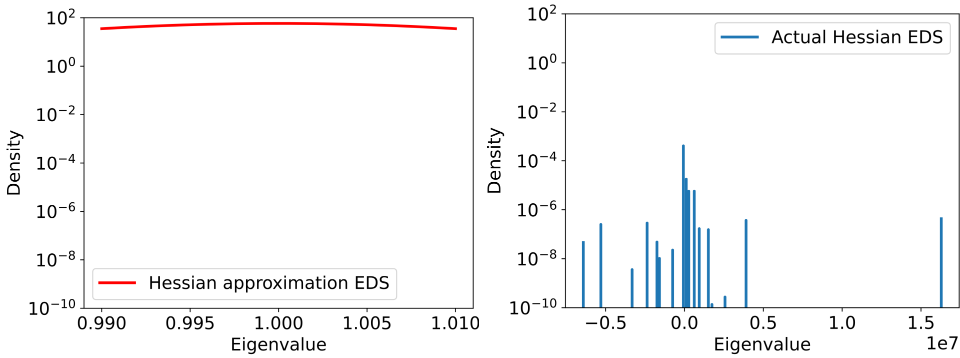


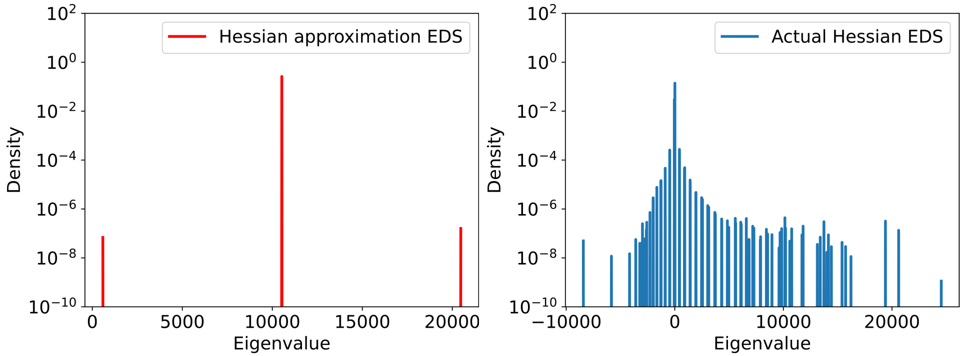
Figure 3.6: Loss curves comparing the cases when the hyper-parameter of L-BFGS is varied when tested on the large problems. *LBFGS_M1* denotes the case with $m = 1$ while *LBFGS* is the default case with $m = 10$.

as such. The second version is L-BFGS without line-search. This is achieved by implementing a recursion-based L-BFGS algorithm [23] that would perform an approximation to the Hessian and compute the step direction by preconditioning the gradient, but without performing the line-search and instead the step size is kept constant (similar to GD). The step size was chosen for each problem using hyper-parameter optimization, as given in section B.7. This derived optimizer is called here *Hessian descent*. The details of the implementation are given in section B.2.

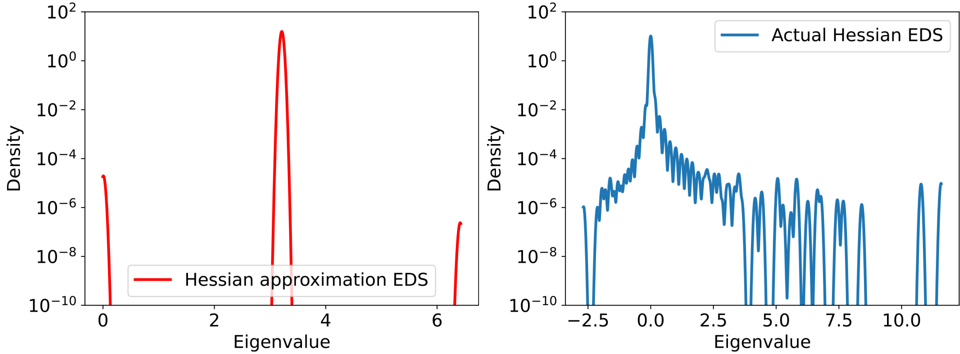
The loss curves obtained by optimizing the test cases using these two optimizers are shown in Figure 3.8 and Figure C.6. Hessian descent was tested with two



(a) At initialization (b) At initialization



(c) After 10 steps (d) After 10 steps



(e) Near convergence (Step = 28) (f) Near convergence (Step = 28)

Figure 3.7: Comparison of EDS of the actual Hessian (right) with that of the approximation (left) made by *LBFGS_M1* for CNNModel, p=3 and problem E for a particular run.

different values of the m hyper-parameter. For the case of $p = 1$, in both models the performances of all these algorithms are very similar. All of them converge to

minima of similar quality in similar number of steps. When considering the case

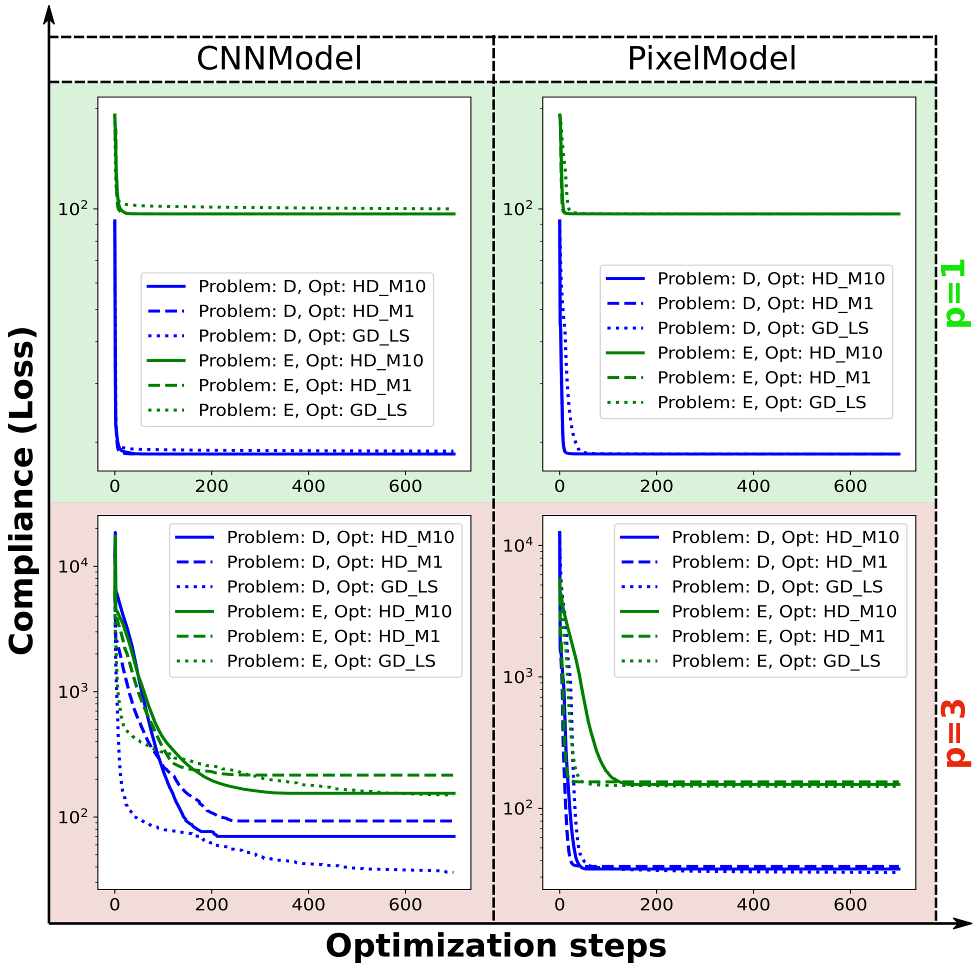


Figure 3.8: Loss curves for Hessian Descent (*HD*) and Gradient Descent with line search (*GD_LS*) for the large problems. The subtexts of *M1* and *M10* correspond to the cases when the hyper-parameter m is set to 1 and 10 respectively.

of *p*=3, the behaviour becomes quite interesting. For the CNNModel, there is a clear difference in the performances of these optimizers. The first observation is that gradient descent with line-search (*GD_LS*) seems to perform either equal to or better than Hessian descent (*HD*) in terms of the attained minima quality. Convergence of this optimizer also seems to be better for most cases. This can also be seen from Table 3.3. The second observation is that that a lower m parameter leads to worse performance for *HD*, i.e. for L-BFGS without line-search. As for the PixelModel, all 3 optimizers seem to perform equally well. In fact, all these methods, which are subsets of the L-BFGS algorithm, seem to perform better than

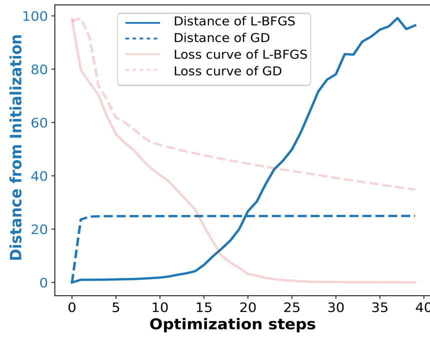
L-BFGS. Further, unlike the case with CNNModel, the performance of *HD* is not hampered with the poor Hessian approximation. On the contrary, Hessian descent with $m = 1$ (*HD_M1*) seems to converge in fewer steps than others (see Figure C.7 and Figure C.8). The reason for this observation in the pixel space is not clear to the authors.

Model	Penal	Problem	GD	HD_M1	HD_M10	GD_LS	LBFGS_M1	L-BFGS
CNNModel	*p=3*	A	79.67 (573)	362.93 (215)	113.55 (473)	53.6 (140)	54.36 (23)	54.61 (22)
		B	13.62 (522)	24.38 (149)	9.11 (452)	8.57 (154)	8.52 (25)	8.54 (24)
		D	61.36 (586)	93.05 (230)	70.11 (204)	36.15 (588)	31.46 (35)	30.68 (36)
		E	275.93 (549)	215.96 (205)	154.77 (286)	149.34 (548)	147.67 (29)	148.47 (29)
	p=1	A	21.29 (6)	21.27 (10)	21.27 (10)	21.6 (6)	21.28 (6)	21.27 (6)
		B	6.14 (2)	6.13 (4)	6.13 (5)	6.29 (16)	6.13 (18)	6.13 (14)
		D	18.47 (4)	18.43 (5)	18.43 (8)	18.81 (6)	18.46 (8)	18.44 (9)
		E	96.79 (6)	96.64 (6)	96.63 (9)	100.03 (9)	96.72 (25)	96.64 (23)
PixelModel	*p=3*	A	89.49 (531)	68.23 (37)	67.8 (46)	67.56 (172)	74.94 (117)	76.12 (181)
		B	9.88 (432)	8.98 (18)	9.2 (24)	8.77 (61)	8.98 (31)	9.16 (23)
		D	46.53 (536)	36.19 (25)	34.55 (42)	32.52 (139)	44.57 (39)	39.72 (31)
		E	175.36 (463)	158.99 (19)	152 (122)	146.74 (57)	168.55 (19)	169.07 (19)
	p=1	A	21.46 (25)	21.27 (6)	21.27 (6)	21.28 (18)	21.27 (7)	21.27 (7)
		B	6.33 (72)	6.13 (11)	6.13 (9)	6.13 (26)	6.13 (9)	6.13 (9)
		D	20.67 (424)	18.44 (8)	18.44 (8)	18.44 (37)	18.44 (11)	18.44 (10)
		E	106.77 (172)	96.63 (5)	96.63 (6)	96.65 (17)	96.63 (6)	96.63 (6)

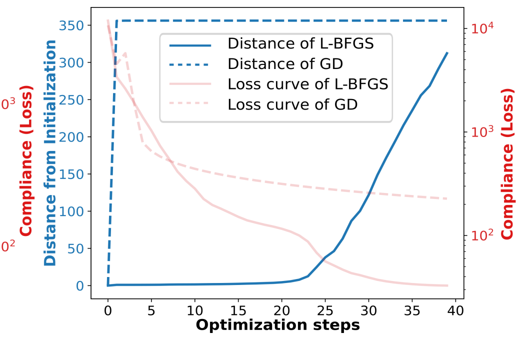
Table 3.3: Summary of the loss obtained on testing the different optimizers on the test combinations. The approximate step at which the optimizer converged is given in the brackets. The loss values represented are the median of the final losses over 10 random seeds.

The importance of line-search for the CNNModel is corroborated by these results. Performing a well tuned descent using only Hessian information produced poor results. The poor performance was further exacerbated when only a single BFGS update was performed which is in clear contrast to what was observed in Figure 3.6 and Figure C.5. This does not mean that the Hessian information is to be completely disregarded because in the case of gradient descent with line-search (*GD_LS*), even though it reached the same minima quality as L-BFGS, it did so in a large number of iterations. This indicates that a rough scaling of the gradient, as indicated by the simple EDS of the approximation matrix, is certainly necessary in these settings for fast convergence. This relative importance of the two components is not trivial, as observed in the case of PixelModel for *p*=3. Even though both gradient descent with line-search (*GD_LS*) and Hessian descent (*HD*) managed to produce better results than their parent L-BFGS, *HD* managed to outperform *GD_LS* in terms of convergence, as shown in Figure C.7 and Figure C.8.

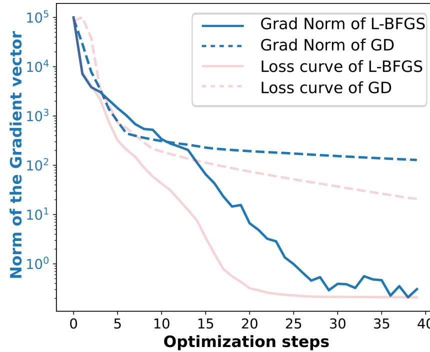
The reason for GD's failure in the CNNModel is also explainable from the results obtained, as good minima are in flat regions of the loss landscapes. "Flat" here is not implying a complete lack of gradient information, rather regions with much weaker gradients than earlier stages of optimization. This is clearly observed in Figure 3.9 where the gradient varies over orders of magnitude. Since a substantial amount of loss reduction happens over these flat lands, GD which is tuned to a



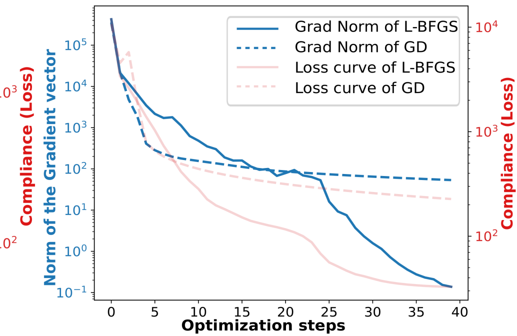
(a) Problem A, p=3



(b) Problem D, p=3



(c) Problem A, p=3



(d) Problem D, p=3

Figure 3.9: Distance moved from initialization and the *L2* norm of the gradient (median over 3 seeds) of L-BFGS and GD for the CNNModel. The loss curves (median over 10 seeds, in red) is shown in reduced opaqueness to show the correlations between the reduction in loss, gradient norm values and the distance moved. Results are shown for problems A and D for $p = 3$ and the rest of the problems exhibit similar patterns, as shown in Appendix C.

fixed learning rate meant for the starting locations is unable to move sufficiently in regions of smaller gradients. L-BFGS on the other hand, by virtue of its line-search, can have very small steps in the beginning, where the landscape is highly non-convex and can also adapt to the flat landscape where larger steps have to be made.

Conclusions

This work provided insights into the optimization process of neurally reparameterized topology optimization problems. The comparative exploration of objective (loss) landscapes for convex and non-convex structural compliance optimization problems revealed that optimizers including line-search reach much better solutions (lower compliance) in fewer iterations. In addition, this investigation also

revealed that if the optimizer includes an approximation of the Hessian matrix (second derivatives) it reaches a similar solution but in fewer iterations. However, the quality of the Hessian approximation used by the optimizer did not seem to be relevant, at least in the case of the L-BFGS algorithm. It can be conjectured that a mere scaling of the gradients done by such matrix is all that is necessary to accelerate convergence because towards the end of the optimization history the optimizer is in basin regions with low gradients. Contrary to deep learning literature, the particular importance of line-search and (to a lesser extent) the effect of including a Hessian approximation justify the superior performance of L-BFGS compared to gradient descent optimizers when using the neural reparameterization trick in topology optimization problems. These conclusions are valid whether the topology optimization problem is convex or non-convex prior to the neural reparameterization.

Furthermore, objective (loss) landscape visualization methods and the Hessian EDS analysis revealed that the reparameterized landscape is far from being convex. Yet, the reparameterization is shown to be beneficial in most topology optimization cases considered, suggesting that optimizers have to explore less when traversing these landscapes and that good solutions are available near the initialization points. This observation is particularly striking for cases with convex objective landscapes prior to reparameterization, since introducing more parameters and non-convexity is shown to still lead to better solutions in similar or fewer iterations.

4

Discussion

In this thesis, the claim made by Hoyer et al [3] that L-BFGS outperforms GD on all problems was confirmed through numerical experiments. With this information and the initial intuition that the main difference between L-BFGS and GD is that the former employs curvature information, we set out to estimate the Hessian and its properties for answers. From the DL literature, we could observe that Hessian-based investigations of neural networks were also on the rise. This gave us the confidence to implement the tools necessary for estimating the enormous Hessian matrix of deep neural networks by making the entire computational graph twice-differentiable. This proved to be a daunting task that demanded an in-depth knowledge into the working of the Python modules Tensorflow [56] and Autograd, which were used to construct the neural part and the finite element analysis parts respectively.

Once the graph was ready, the two tools necessary for estimating the Hessian were adapted to the current settings from open-source code. The implementation was checked thoroughly by conducting experiments on smaller models where the Hessian could be calculated exactly. Once the results obtained from estimating the Hessian and exactly calculating it were found to match, further experimentation was possible. The other tools for visualizing the landscape and analyzing the trajectories were much simpler and straight forward to implement.

The first experiment was to test the claim made by Hoyer et al [3] by comparing L-BFGS and GD on all the selected test cases. The linear interpolation tool revealed a smooth slice of the landscape that seemed to be convex, albeit with slight bumps for some cases. Our initial thoughts were that if the landscape was indeed convex, that would certainly explain why L-BFGS outperforms GD. But these visualization tools are extremely low dimensional and interpretations about the much higher dimensional loss landscape should be made with caution. The 2D projection onto filter-normalized random vectors and the Hessian EDS revealed that the landscape

is far from being convex. This suggested the possibility that the optimizers have to explore less and that good solutions were available near the initializations.

Still far from the answers we sought, we decided to study the algorithms of the optimizers in greater detail. From this we understood that L-BFGS does indeed make an estimation of the Hessian and uses that to find the step direction. But it also uses a line-search algorithm to find the correct step size at each iteration. Now, we had two ideas to follow; the first was to compare the L-BFGS's approximation of the Hessian with the actual one and the second was to isolate the two components of L-BFGS and to study them apart.

Estimating the EDS of the approximation required a way to compute the "Approximation-vector" product similar to the "Hessian-vector" product oracle implemented for the Hessian EDS. Since the L-BFGS code was in Fortran, we had to implement the necessary parts in Python. The Hessian EDS and the approximation's EDS were then compared for L-BFGS and although it was not a good match, it was difficult to be certain about it. In order to have an easier comparison, we decided to limit the BFGS updates of L-BFGS to just one by changing the m hyper-parameter. This formed the second experiment and to our surprise, this optimizer (*LBFGS_M1*) had the same performance as the original L-BFGS with 10 BFGS corrections. The comparison of the Hessian spectra was much easier in this case and it was clear that L-BFGS does not need to make a good approximation to perform well. Perhaps, a simple scaling of the gradient is all that is needed from this matrix.

After that, we managed to isolate the components of L-BFGS into 2 optimizers; the Hessian descent (*HD*) optimizer that performs no line-search and the gradient descent with line-search (*GD_LS*) which does not perform an approximation to the Hessian. We tested both these optimizers in experiment-3 and found some interesting observations. Firstly, in the neural landscape, HD performed worse than GD_LS on all the problems. Further, when the number of BFGS updates was kept as one for HD, its performance degraded more. GD_LS managed to reach similar minima quality as L-BFGS but the convergence was slow. This seems to suggest that line-search is really important for the neural landscape in order to adapt the sizes to this difficult landscape but the gradient scaling, even by a poor approximation of the Hessian speeds this process up greatly.

4.1. Limitations and future research

For every scientific research, it is important to highlight the important limitations. Emphasizing on the failures not only provides assistance to future investigators but also paves the path for diverse research directions.

- *Properties of the Hessian, as chosen in the literature, were uninformative in differentiating good initializations.* A lot of effort was involved in constructing estimates to these properties with the aim to predict the quality of final minima from a few steps of the optimizer. This is understandable since L-BFGS was shown to not require a quality approximation of the Hessian and even choosing m to be one resulted in equal performance. Thus, L-BFGS needs only local

information about the landscape to reach the minima and extracting global information from its iterates is difficult. Although we could not extract such information, perhaps an application of Machine Learning might uncover the hidden trends.

- *Testing SGD on neural reparametrization*. It can be argued that SGD would provide a better baseline than GD, firstly due to its ubiquitous use in Deep Learning and more importantly, because of the hypothesis that the noise in SGD enhances its performance [24, 40, 59]. In this thesis, due to the lack of stochasticity in the adopted framework, the focus on optimization and the need for an easy to compare baseline, this is not needed. But by adding noise of different structures and magnitudes to the gradient calculation, it is possible to further study this hypothesis.
- *Performing PCA on iterates can be misleading*. Projecting the landscape around the minima using PCA is a visualization tool available in DL literature (see **??**). Li et al ([42]) who introduced this technique remark that the optimization trajectory is extremely low dimensional. This observation was due to the total variance explained by the first two components being close to 100%. Although, it may be true that the optimization subspace is low dimensional in comparison to the total number of parameters, these numbers can give the wrong interpretation of the trajectory. In the current framework, this total explained variance was computed at each iterate and it was observed that the trajectory is not 1D or 2D in the early stages and the total variance of the two components do not add up to 100%. The disguise of a 1D or 2D trajectory stems from the large steps taken near the minima in comparison to the small steps in the beginning of optimization. These later steps are so large that they mask the influence of the early steps (See Figure 3.9). It is to be seen whether this effect is also responsible for the apparent low dimensionality of the trajectories in DL applications as well.
- *Testing GD with a step size schedule and different preconditioners*. Since we observed that the line-search helps in having smaller step sizes at the beginning and large ones near the minima in a smooth transition, it can be expected that a simple and increasing step size schedule may also result in better solutions. This could be computationally advantageous in problems where the line-search performs multiple objective function evaluations to choose a step size. Further, rescaling the gradients using simple matrices, due to the simple structure of the Hessian approximation made by L-BFGS, alongside this step size scheduling may result in a better optimizer.

References

[1] E. Andreassen, A. Clausen, M. Schevenels, B. Lazarov, and O. Sigmund, *Efficient topology optimization in matlab using 88 lines of code,* Structural and Multidisciplinary Optimization **43**, 1 (2011).

[2] T. Hey, K. Butler, S. Jackson, and J. Thiyagalingam, *Machine learning and big scientific data,* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **378**, 20190054 (2020), https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2019.0054 .

[3] S. Hoyer, J. Sohl-Dickstein, and S. Greydanus, *Neural reparameterization improves structural optimization,* (2019).

[4] A. Chandrasekhar and K. Suresh, *Tounn: Topology optimization using neural networks,* Structural and Multidisciplinary Optimization **63**, 1 (2021).

[5] IIija Mihajlovic, *Everything you ever wanted to know about computer vision.* (2019), [Online; accessed October 20, 2021].

[6] D. Ulyanov, A. Vedaldi, and V. Lempitsky, *Deep image prior,* 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition , 9446 (2018).

[7] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, Cambridge, MA, USA, 2016) `http://www.deeplearningbook.org`.

[8] I. Sosnovik and I. Oseledets, *Neural networks for topology optimization,* Russian Journal of Numerical Analysis and Mathematical Modelling **34**, 215 (2017).

[9] Y. Zhang, A. Chen, B. Peng, X. Zhou, and D. Wang, *A deep convolutional neural network for topology optimization with strong generalization ability,* ArXiv **abs/1901.07761** (2019).

[10] X. Lei, C. Liu, Z. Du, W. Zhang, and X. Guo, *Machine learning driven real time topology optimization under moving morphable component (mmc)-based framework,* Journal of Applied Mechanics **86** (2019), 10.1115/1.4041319.

[11] E. Ulu, R. Zhang, and L. B. Kara, *A data-driven investigation and estimation of optimal topologies under variable loading configurations,* Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization **4**, 61 (2016), https://doi.org/10.1080/21681163.2015.1030775 .

[12] T. Guo, D. Lohan, R. Cang, Y. Ren, and J. Allison, *An indirect design representation for topology optimization using variational autoencoder and style transfer,* (2018).

[13] Y. Yu, T. Hur, J. Jung, and I. Jang, *Deep learning for determining a near-optimal topological design without any iteration,* Structural and Multidisciplinary Optimization **59**, 787 (2018).

[14] Z. Nie, T. Lin, H. Jiang, and L. Kara, *Topologygan: Topology optimization using generative adversarial networks based on physical fields over the initial domain,* ArXiv **abs/2003.04685** (2020).

[15] O. Sigmund and K. Maute, *Topology optimization approaches a comparative review,* Structural and Multidisciplinary Optimization **48** (2013), 10.1007/s00158-013-0978-6.

[16] A. Mordvintsev, N. Pezzotti, L. Schubert, and C. Olah, *Differentiable image parameterizations,* Distill (2018), 10.23915/distill.00012, https://distill.pub/2018/differentiable-parameterizations.

[17] D. Maclaurin, D. Duvenaud, and R. P. Adams, *Autograd: Effortless gradients in numpy,* in *ICML 2015 AutoML Workshop*, Vol. 238 (2015) p. 5.

[18] L. Bottou, F. E. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning,* (2016).

[19] C. Li, H. Farkhoor, R. Liu, and J. Yosinski, *Measuring the intrinsic dimension of objective landscapes,* (2018).

[20] E. Weinan, C. Ma, S. Wojtowytsch, and L. Wu, *Towards a mathematical understanding of neural network-based machine learning: what we know and what we don't,* ArXiv **abs/2009.10713** (2020).

[21] W. E, C. Ma, S. Wojtowytsch, and L. Wu, *Towards a mathematical understanding of neural network-based machine learning: what we know and what we don't,* (2020).

[22] Y. Zhou, J. Yang, H. Zhang, Y. Liang, and V. Tarokh, *Sgd converges to global minimum in deep learning via star-convex path,* (2019).

[23] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. (Springer, New York, NY, USA, 2006).

[24] Z. Allen-Zhu, *Natasha 2: Faster non-convex optimization than sgd,* in *NeurIPS* (2018).

[25] A. Mokhtari and A. Ribeiro, *Global convergence of online limited memory bfgs,* (2014), arXiv:1409.2045 .

[26] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. W. Mahoney, *Adahessian: An adaptive second order optimizer for machine learning,* (2020).

[27] D. Goldfarb, Y. Ren, and A. Bahamou, *Practical quasi-newton methods for training deep neural networks,* ArXiv **abs/2006.08877** (2020).

[28] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, *On optimization methods for deep learning,* in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11 (Omnipress, Madison, WI, USA, 2011) p. 265–272.

[29] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, *Large scale distributed deep networks,* in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12 (Curran Associates Inc., Red Hook, NY, USA, 2012) p. 1223–1231.

[30] V. Ramamurthy, *L-sr 1 : A novel second order optimization method for deep learning,* (2016).

[31] K. Li and J. Malik, *Learning to optimize,* ArXiv **abs/1606.01885** (2017).

[32] Y. Dauphin, R. Pascanu, Çaglar Gülçehre, K. Cho, S. Ganguli, and Y. Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization,* in *NIPS* (2014).

[33] Y. LeCun and C. Cortes, *MNIST handwritten digit database,* (2010).

[34] A. Krizhevsky, V. Nair, and G. Hinton, *Cifar-10 (canadian institute for advanced research),* .

[35] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition,* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) , 770 (2016).

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition,* Proceedings of the IEEE **86**, 2278 (1998).

[37] R. Y. Sun, *Optimization for deep learning: An overview,* Journal of the Operations Research Society of China **8**, 249 (2020).

[38] I. J. Goodfellow, O. Vinyals, and A. M. Saxe, *Qualitatively characterizing neural network optimization problems,* (2014).

[39] J. Frankle, *Revisiting qualitatively characterizing neural network optimization problems,* (2020).

[40] C. Xing, D. Arpit, C. Tsirigotis, and Y. Bengio, *A walk with sgd,* (2018).

[41] J. Lucas, J. Bae, M. R. Zhang, S. Fort, R. Zemel, and R. Grosse, *Analyzing monotonic linear interpolation in neural network loss landscapes,* (2021).

[42] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, *Visualizing the loss landscape of neural nets,* (2017).

[43] A. Chatzimichailidis, F.-J. Pfreundt, N. R. Gauger, and J. Keuper, *Gradvis: Visualization and second order analysis of optimization surfaces during the training of deep neural networks,* (2019).

[44] Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney, *Pyhessian: Neural networks through the lens of the hessian,* (2019).

[45] S. Jastrzebski, M. Szymczak, S. Fort, D. Arpit, J. Tabor, K. Cho, and K. Geras, *The break-even point on optimization trajectories of deep neural networks,* (2020).

[46] B. Ghorbani, S. Krishnan, and Y. Xiao, *An investigation into neural net optimization via hessian eigenvalue density,* (2019).

[47] G. Gur-Ari, D. A. Roberts, and E. Dyer, *Gradient descent happens in a tiny subspace,* (2018).

[48] B. A. Pearlmutter, *Fast exact multiplication by the hessian,* Neural Comput. **6**, 147–160 (1994).

[49] L. Wu, Z. Zhu, and W. E, *Towards understanding generalization of deep learning: Perspective of loss landscapes,* (2017).

[50] G. Alain, N. L. Roux, and P.-A. Manzagol, *Negative eigenvalues of the hessian in deep neural networks,* (2019).

[51] S. Fort and S. Ganguli, *Emergent properties of the local geometry of neural loss landscapes,* (2019).

[52] S. Ubaru, J. Chen, and Y. Saad, *Fast estimation of tr(f(a)) via stochastic lanczos quadrature,* SIAM Journal on Matrix Analysis and Applications **38**, 1075 (2017).

[53] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou, *Empirical analysis of the hessian of over-parametrized neural networks,* (2017).

[54] R. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization,* SIAM J. Sci. Comput. **16**, 1190 (1995).

[55] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,* Nature Methods **17**, 261 (2020).

[56] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems,* (2015), software available from tensorflow.org.

[57] T. Garipov, P. Izmailov, D. Podoprikhin, D. Vetrov, and A. G. Wilson, *Loss surfaces, mode connectivity, and fast ensembling of dnns,* (2018).

[58] F. Draxler, K. Veschgini, M. Salmhofer, and F. A. Hamprecht, *Essentially no barriers in neural network energy landscape,* (2018).

[59] R. D. Kleinberg, Y. Li, and Y. Yuan, *An alternative view: When does sgd escape local minima?* in *ICML* (2018).

[60] I. Safran and O. Shamir, *On the quality of the initial basin in overspecified neural networks,* (2015).

[61] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization,* (2014).

[62] E. Littwin and L. Wolf, *On the convex behavior of deep neural networks in relation to the layers' width,* (2020).

[63] R. Kleinberg, Y. Li, and Y. Yuan, *An alternative view: When does sgd escape local minima?* (2018).

[64] Q. Yuan and N. Xiao, *Experimental exploration on loss surface of deep neural network,* International Journal of Imaging Systems and Technology **30**, 860 (2020).

[65] A. Gotmare, N. S. Keskar, C. Xiong, and R. Socher, *Using mode connectivity for loss landscape analysis,* (2018).

[66] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin, *Linear mode connectivity and the lottery ticket hypothesis,* (2019).

[67] S. Fort, G. K. Dziugaite, M. Paul, S. Kharaghani, D. M. Roy, and S. Ganguli, *Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the neural tangent kernel,* (2020).

[68] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,* Nature Methods **17**, 261 (2020).

[69] R. Byrd, J. Nocedal, and B. Schnabel, *Representations of quasi-newton matrices and their use in limited memory methods,* Mathematical Programming **63**, 129 (1994).

[70] J. Bergstra, D. Yamins, and D. D. Cox, *Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures,* in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13 (JMLR.org, 2013) p. I–115–I–123.

A

Supplementary information for Chapter - 2

A.1. BFGS algorithm details

Since the L-BFGS algorithm is derived from BFGS, it is logical to first understand how BFGS works. As mentioned before, Quasi-newton methods make an approximation to the actual Hessian and uses it in place of the second-derivative information to form the quadratic model in Equation 2.5. In the case of BFGS, this is achieved entirely using only first-order information about the objective landscape. Intuitively, it is easy to imagine that the change in gradients from one step to another carries information about the curvature. The BFGS algorithm tries to capture this over multiple steps to create better approximations to the actual curvature. If at an iterate k, the quadratic model built by the BFGS algorithm is given by:

$$m_k(\mathbf{p}) = f_k + \vec{\nabla} f_k^T \mathbf{p} + \frac{1}{2}\mathbf{p}^T \mathbf{B}_k \mathbf{p} \tag{A.1}$$

Equation A.1 is just a rearrangement of the Taylor Equation (2.5) where the matrix $\mathbf{B_k}$ represents the current approximation to the Hessian. In order to update $\mathbf{B}_k$ to $\mathbf{B}_{k+1}$, certain conditions are imposed, the first of which is that the model built using $\mathbf{B}_{k+1}$, should have the same gradient as the actual objective function at $\mathbf{x}_{k+1}$ (this is automatically satisfied). The second condition is that the new approximation ($\mathbf{B}_{k+1}$) ensures that the model's gradient at k matches as well. In short, the minimum requirement of an update is that the model approximates the objective function, to first-order, at both the new and previous iterates. By imposing these conditions, the secant condition is derived and is expressed as:

$$\mathbf{B}_{k+1}\mathbf{s}_k = \mathbf{y}_k \tag{A.2}$$

where $\mathbf{s}_k$ is the step vector given by $\mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k$ is the change in gradients given by $\vec{\nabla} f_{k+1} - \vec{\nabla} f_k$. Adding the extra information that Hessian is symmetric

A

and that successive updates should be close to each other to the secant condition, the BFGS update equation is obtained. Since we require the inverse of the Hessian to find the search direction, for the ease of implementation, these conditions are instead imposed on the inverse of $\mathbf{B}$ matrix i.e. on $\mathbf{M} = \mathbf{B}^{-1}$. This procedure is described in *Algorithm 2* and is a restatement of the same given in [23]. Note that the algorithm never forms the full inverse Hessian matrix explicitly, instead only the vector pairs $(\mathbf{s}_k, \mathbf{y}_k)$ are stored. All that is required is its product with a given vector (here, the gradient). There are computationally efficient ways to achieve this. See the Pearlmutter's trick [48] for more details.

Algorithm 2: BFGS algorithm

Given: initialization point $\mathbf{x}_0$, tolerance ϵ, starting Hessian matrix $\mathbf{M}_0$;
k ← 0;
while $\left|\vec{\nabla} f_k\right| > \epsilon$ **do**

Compute search direction with current approximation;

$$\mathbf{p}_k = -\mathbf{M}_k \vec{\nabla} f_k \tag{A.3}$$

Compute next iterate;

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{A.4}$$

where α_k is the step size in the chosen direction and is positive scalar obtained using a line search algorithm ;
Update the approximation using new information;

$$\mathbf{M}_{k+1} = (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T)\mathbf{M}_k(\mathbf{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T \tag{A.5}$$

where $\rho_k = \frac{1}{\mathbf{y}_k^T \mathbf{s}_k}$ is a scalar. ;
k ← k+1

end

A.2. Observed empirical properties of neural landscapes

The literature on the loss landscapes in Deep Learning applications have shown that these landscapes possesses particular properties. These properties are highlighted below.

A.2.1. Quasi-convexity

An important property that is prevalent in the literature is a sort of quasi-convexity for the training of neural networks in relation to their optimization trajectories. There are many hints at this property that can be observed across multiple papers.

Firstly, it is the observation by Goodfellow et al [38] that the loss is monotonically

decreasing from the initialization point to the converged minima when the 1D projection method is used. They hypothesized that despite the non-convex nature of general NN loss landscapes, this Monotonic Linear Interpolation (MLI) property implied some level of convexity in the sense that, if the direction of minima was known, a coarse line search would be enough to find the minima.
Using this method, Frankle et al [39] found that this MLI property is not valid in general for more complex datasets and architectures, as shown in Figure 2.9. They also validated the results of [38] for MNIST and observed that the MLI property was valid for any iterate of the optimizer as well. So the observation that NNs are easy to optimize because of the lack of apparent non-convexity in the linear subspace seems to work only for simplified settings where the optimization task is relatively easier. One caveat is that [39] plotted the test loss while [38] plotted both training loss and test loss. Safran et al [60], by studying feedforward networks with a convex loss function, also suggested that this property may be indicative of the difficulty of the optimization task, in line with the observations of [38].

The MLI property was studied recently by Lucas et al [41]. They found that this was a global property of the landscape because once an initialization-minima pair had MLI, it existed for any other random point (from the same initialization scheme) and the said minima. Further, it existed between pairs of initializations as well. They theoretically and then through experiments showed MLI could be broken by using large learning rates for SGD, using adaptive optimizers like ADAM [61] or by using batch normalization. Intuitively, if the weights move further from the initial point, there is a higher probability that the optimizer can jump from one convex basin to another. They hypothesized that since the networks with MLI move less, they can be approximated to be linear along this subspace. This is understood to mean that the quadratic approximation at the minima becomes valid for most initialization points and the optimizer simply falls into the nearest basin, as in convex optimization. In the appendix to the main paper, they proved that MLI property always exists for convex landscapes.

Secondly, it is the observation by Guy et al [47] that the optimization happens in a tiny subspace for gradient descent (GD). This tiny subspace was found to be the eigenspace formed by the top eigenvectors of the Hessian which were all positive. And since GD was restricted to be this subspace of purely positive curvature, a sort of convex behaviour is to be expected. The low-D nature of the optimization trajectory was also corroborated in [42] because most of the variance in the iterates were explained using only the first two principal components of the PCA, when plotting 2D projections. The study by Littwin et al [62] also lends credence to this property because they observed that the curvature along the gradients at each iterate is strictly positive which is especially pronounced for wider networks.

Finally, Zhou et al [22] and Kleinberg et al [63] both define a property called star-convexity of the optimization trajectory of SGD which ensures that the negative of the gradient always point towards the minima at every iterate after an early period. This was tracked using a metric called the residual $\epsilon = \mathcal{L}(\vec{\theta}_i) - \mathcal{L}(\vec{\theta}_T) +$

$\left\langle \vec{\theta}_T - \vec{\theta}_i, \nabla \mathcal{L}(\vec{\theta}_i) \right\rangle$ which would always be negative if this property was valid. Kleinberg et al [63] proved a similar result empirically not just for the iterates but also for randomly sampled points in the neighbourhood of the iterates of the optimizer, where the neighbourhood radius was chosen as the maximum step size taken. In order to test whether this property was valid for other minima as well, they fixed the first 10 epochs of training and then generated 50 distinct minimas using SGD. It was found that for ResNet and DenseNet on CIFAR-10, each iterate of every trajectory has this property with respect to all the minimas, seemingly suggesting that the loss surface is skewed towards the minima globally and locally. But all these findings for SGD seems to indicate a benevolent global and local loss landscape for most of the SOTA networks and datasets.

A.2.2. Mode connectivity and predictability

Garipov et al [57] and Draxler et al [58] independently found an interesting property of the minima of NN loss landscapes. They found that for overparameterized NNs, which include most of the SOTA networks, the minima or modes as they call them, can be connected by simple low-loss non-linear pathways. They termed this property as mode connectivity and found a way to connect any two independently trained NN minimas on popular datasets and network architectures. This would indicate that the notion of an isolated minima, as is the case in lower dimensions, is no longer applicable to such overparameterized regimes. Instead, the minimas of NN loss landscapes are to be thought of as basins. Further, these basins are very high-D with multiple redundant directions, as evidenced by the bulk of eigenvalues of the Hessian of the loss being near zero at the minimas [53].

Apart from this empirical observation about the Hessian, Li et al [19] introduced the concept of an intrinsic dimension of objective landscapes and showed that the number of parameters of the network above a certain task dependent intrinsic dimension d_{int}, contributes only to the redundancy of the solution set. They also showed that d_{int} is related to the difficulty of optimization of a particular task(e.g. for a Fully Connected NN for classifying digits from the MNIST dataset, the value was about 750, when the the network had $\approx$ 200,000 parameters). These findings were corroborated and further extended in [51], where it was showed that even training on a very low-D random hyper-plane or hyper-sphere embedded in the high-D space, results in similar performance to full space training, on popular datasets and networks. They introduced the concept of a Goldilocks zone in NN loss landscapes, a region of higher than average positive curvature as measured by certain Hessian metrics, which accounted for this observation. Many authors have proposed ensembling the mode connector curves to improve the capabilities of a NN model [57], [64].

This mode connectivity (MC) property was used to analyze the loss landscapes in [65] and [66]. Gotmare et al [65] firstly showed that the method of finding the MC curve was robust to the choice of optimizer, initialization and other optimizer hyper-parameters i.e. an MC curve could be formed between two minimas found by varying any of these hyper-parameters. An example of such a curve is shown

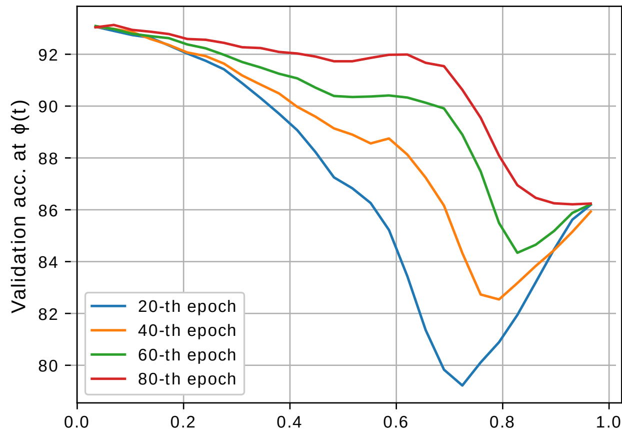


Figure A.1: The MC curve between the minima found by Adam optimizer and the one found by SGD. The MC curve is parameterized by $t \in [0, 1]$. The algorithm for finding the curve $\phi(t)$ is iterative and the curve obtained at different progress steps is also shown [65]

in Figure A.1. They then chose to analyze SGD with warm restarts by constructing both linear interpolation curves and MC curves between certain iterates and contrasting the observations to regular SGD. They also formed a plane using 2 iterates and the best point on the MC curve and by projecting the other iterates onto this plane showed that SGD with warm restarts does not jump from one minima to other, refuting the popular claim. One particular use of these curves in Hoyer et al [3] would be to connect two final minima and find other designs with lower compliance values than the connected two.

Frankle et al [66], on the other hand, found that when a parent network is trained for k iterations ($k \ll$ total iterations for convergence) and it spawns child networks which are trained with different hyper-parameters of the optimizer, the final modes attained are all linearly connected i.e. there is no loss barrier between the linear interpolation between the minimas of the child network. This property was termed as linear mode connectivity and was valid for the popular datasets and common network architectures for the SGD optimizer (k ≈ 3% of the total steps for ResNet-20 on CIFAR-10 while it was ≈ 0 on MNIST for common networks). This seems to suggest that the final minima basin of an optimizer on a loss landscape, even for a stochastic optimizer like SGD, is predetermined at an extremely early stage in the training even for the most difficult optimization tasks. This is in line with most of the current evidence that seems to suggest that there are two regimes of training. The first is the chaotic and short regime, which determines the final basin of minima to which the optimizer will converge to and the second is the more stable regime of optimization [67]. So, determination of final minima quality may be possible much early in the training process. Also since the MC represents global information that connects multiple regions of the objective landscape, this could be a valuable

tool.

B

Supplementary information for chapter-3

B.1. Computational graph details

This section will try to include the details about the various codes that have been used in this thesis.

B.1.1. Basic framework

The basic computational framework was adopted from the code made available by Hoyer et al [3]. This involved the set of TO problems, the FEM model for calculating the compliance (Written in Autograd numpy) and the Neural architecture (Written in Tensorflow 2 [56]). Everything was built on top of this framework. However, a few important changes were made and these will be detailed below. Whatever is omitted can be safely assumed to be part of the original code.

B.1.2. Twice differentiable graph

The architecture that was chosen for the experiments in [3] was a Convolutional Neural Network (CNN) that employed both Convolutional layers and Upsampling layers to produce the structural image. But in order to utilize the tests on the Hessian, the second-order information has to flow in the computational graph and this required a twice-differentiable computational graph. The original architecture did not support this and thus a similar architecture that used Transposed Convolution layers instead of the convolution + upsampling combination was employed in all of the experiments performed in this thesis. This was achieved through the use of *Conv2DTranspose* layer of Keras with the *same* padding option. This resulted in the CNN part being twice differentiable. Another change was to the function that interlinked the CNN with the TO physics. Extra code was written, utilizing the *vjp* capabilities of Autograd and the *custom_gradient* decorators of Tensorflow.

B.1.3. TO problems

The TO problem formulations were also taken from the framework of Hoyer et al [3]. These problems are named in the original paper as:

- **A**- michell_centered_both_32x64_0.12
- **B** - pure_bending_moment_32x64_0.15
- **C** - crane_128x128_0.2
- **D** - causeway_bridge_middle_128x128_0.1

B.2. Optimizer details

This section will include all the necessary details regarding the optimizers that were used in the experiments.

B.2.1. L-BFGS

L-BFGS was run with the default hyper-parameters using the *optimize.fmin_l_bfgs_b* function of Scipy module [55]. The only parameters that were changed were:

- *maxfun* was set to the maximum number of optimization steps
- *factr* was set to 1.0
- *pgtol* was set to 1e-14

B.2.2. Gradient Descent (GD)

GD was run utilizing the function *keras.optimizers.SGD* from Tensorflow [56] where learning rate was the only hyper-parameter varied.

B.2.3. Hessian Descent (HD)

Hessian Descent was derived from L-BFGS by removing line-search and involving only the Hessian approximation capabilities. In essence, the optimizer would perform an approximation to the Hessian using a twin-loop recursion algorithm ([23]) and compute the step direction by acting on the gradient. But line-search will not be performed and instead the step size is kept constant (similar to the GD case). The learning rate/step size was chosen for each problem using hyper-parameter optimization, as given in Appendix B. The pseudo-code for obtaining the step direction for this optimizer is given in algorithm 3 where the symbols have the same meaning as used in subsection 2.3.2 and section A.1.

B.2.4. Gradient Descent with line-search (GD_LS)

GD_LS had its Hessian approximation capabilities completely removed, which would essentially be the case of L-BFGS with "*m*=0". This could be achieved by simply restarting the L-BFGS algorithm, by providing *maxfun* as 1, for each iteration since the first iteration of L-BFGS is always a simple gradient descent with line search. This was run for 700 iterations on all test cases over 10 random seeds for the

Algorithm 3: Two loop recursion [23]

Result: $\mathbf{r} = \mathbf{M}_k \vec{\nabla} f_k$ = Step direction
Given: *m*, gradient = $\vec{\nabla} f_k$, optimization step k ;
$\mathbf{q} \leftarrow \vec{\nabla} f_k$;
for *i = k-1, k-2, ⋯, k-m* **do**
  $\alpha_i \leftarrow \rho_i \mathbf{s}_i^T \mathbf{q}$;
  $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$;
end
$\mathbf{r} \leftarrow \mathbf{H}_k^0 \mathbf{q}$;
for *i = k-m, k-m+1, ⋯, k-1* **do**
  $\beta \leftarrow \rho_i \mathbf{y}_i^T \mathbf{r}$;
  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$;
end

small problems and 6 random seeds for the larger problems. The optimization was performed and the loss values were recorded.

B.3. Hessian EDS for the loss function

In order to confirm whether the modified SIMP approach is in fact convex for $p = 1$, the eigenvalue density spectrum of the Hessian was computed. If it is a convex function, it should have a positive semi-definite Hessian and this is confirmed in Figure B.1 for all the four TO problem formulations. It is to be highlighted that the Hessian was calculated not with respect to the output of the neural network. The volume constraint is applied to the output of the neural network using sigmoid function and a root solver (see [3]). These altered pixels are then passed onto the modified SIMP model. The Hessian was calculated with respect to these altered pixels.

B.4. Computing the approximate Hessian's spectra

It was necessary to compare the eigenvalue density spectrum (EDS) of the actual Hessian with that of the approximation made by L-BFGS at each step of the optimization process. This comparison would reveal whether the success of L-BFGS on these problems is due to its superior approximation of Hessian. The framework to achieve this would be similar to the one discussed for estimating the Hessian EDS, involving a matrix-vector product oracle and the stochastic lanczos quadrature (SLQ) algorithm, where the matrix would be the approximate Hessian matrix. There was no need to modify the SLQ algorithm but in order to obtain the matrix-vector product without explicitly forming the huge matrix, we had to understand the working of the L-BFGS implementation that we used for our experiments.

The algorithm that we have used is the L-BFGS-B from the Scipy module [68] of Python and works on the same principle as the L-BFGS algorithm detailed in subsec-

B

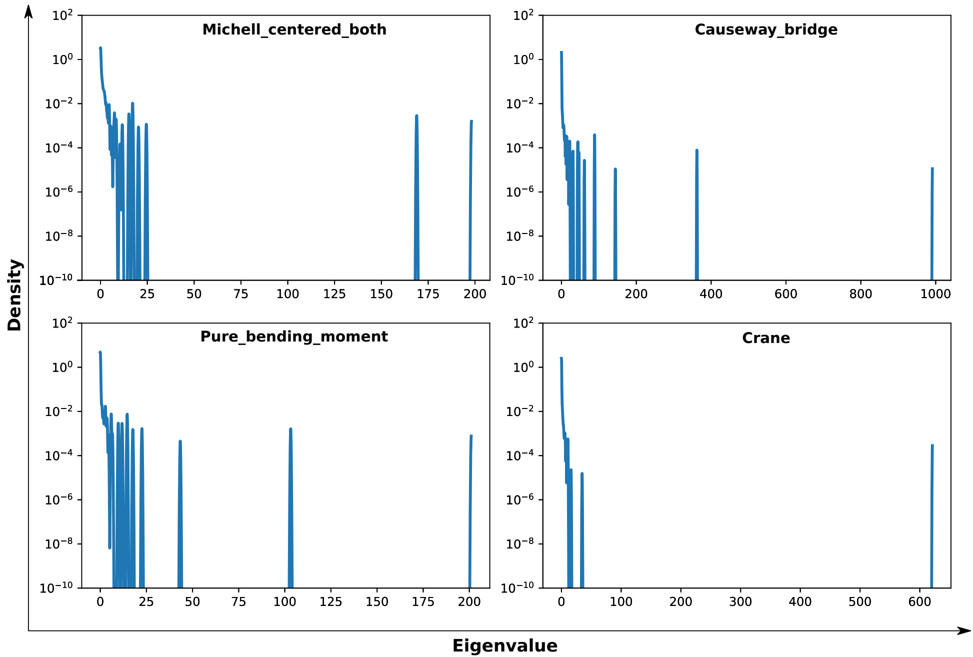


Figure B.1: The Hessian's Eigenvalue Density Spectrum (EDS) for the 4 selected TO problems when the penalization factor p=1 (See section 2.4.2 for details on plotting EDS). This plot captures the positive semi-definiteness of the Hessian which is required for a convex problem formulation. The lowest eigenvalue was numerically verified to be positive as well.

tion 2.3.2. The main difference with respect to the original L-BFGS is that L-BFGS-B is capable of accepting bounds on the variables. This is not the case for us as there are no explicit bounds on the variables. Even though the new algorithm works equally well as the original L-BFGS on unbound problems, the implementation is quite different. It should be noted that only the parts relevant to us will be explained here and for a more detailed account, the reader is advised to visit the paper by Nocedal et al [54].

In the algorithm of L-BFGS-B, there is no two-loop recursion utilizing the **M** matrix to find the step direction. Instead, the Hessian approximation **B** is updated directly using a compact form of the component matrices [69]. The compact representation of such matrices was introduced in [69] and this paper also discusses the most efficient way to compute the product of the **B** matrix with a given vector among other things. The compact form of the BFGS matrix is originally given by the equation:

$$\mathbf{B}_k = \mathbf{B}_0 - \begin{bmatrix} \mathbf{B}_0\mathbf{S}_k & \mathbf{Y}_k \end{bmatrix} \begin{bmatrix} \mathbf{S}_k^T\mathbf{B}_0\mathbf{S}_k & \mathbf{L}_k \\ \mathbf{L}_k^T & -\mathbf{D}_k \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{S}_k^T\mathbf{B}_0 \\ \mathbf{Y}_k^T \end{bmatrix} \tag{B.1}$$

where,

$$\mathbf{S}_k = [\mathbf{s}_{k-m}, \cdots, \mathbf{s}_{k-1}]_{n\times m}, \quad \mathbf{Y}_k = [\mathbf{y}_{k-m}, \cdots, \mathbf{y}_{k-1}]_{n\times m} \tag{B.2}$$

$$\mathbf{B}_0 = \delta_k \mathbf{I} \tag{B.3}$$

$$\mathbf{D}_k = diag[\mathbf{s}_{k-m}^T \mathbf{y}_{k-m}, \cdots, \mathbf{s}_{k-1}^T \mathbf{y}_{k-1}]_{m\times m} \tag{B.4}$$

$$(\mathbf{L}_k)_{i,j} = \begin{cases} (\mathbf{s}_{k-m-1+i}^T)(\mathbf{y}_{k-m-1+j}) & \text{if } i > j \\ 0 & \text{otherwise} \end{cases} \tag{B.5}$$

The matrix $\mathbf{L}_k$ also has the same dimension as the matrix $\mathbf{D}_k$. The above equations are for the general case when $k > m$. If $k < m$, the only change is that the dimensions of the matrices $\mathbf{Y}, \mathbf{S}, \mathbf{D}$ and $\mathbf{L}$ will have only k columns instead of m. In order to have an efficient implementation, the cost of inversion of the $2m \times 2m$ middle matrix in Equation B.1 has to be reduced and is thus replaced by a rearranged matrix which can be easily factorized as shown below.

$$\begin{bmatrix} -\mathbf{D}_k & \mathbf{L}_k^T \\ \mathbf{L}_k & \mathbf{S}_k^T \mathbf{B}_0 \mathbf{S}_k \end{bmatrix} = \begin{bmatrix} \mathbf{D}_k^{0.5} & 0 \\ -\mathbf{L}_k \mathbf{D}_k^{-0.5} & \mathbf{J}_k \end{bmatrix} \begin{bmatrix} -\mathbf{D}_k^{0.5} & \mathbf{D}_k^{-0.5} \mathbf{L}_k^T \\ 0 & \mathbf{J}_k^T \end{bmatrix} \tag{B.6}$$

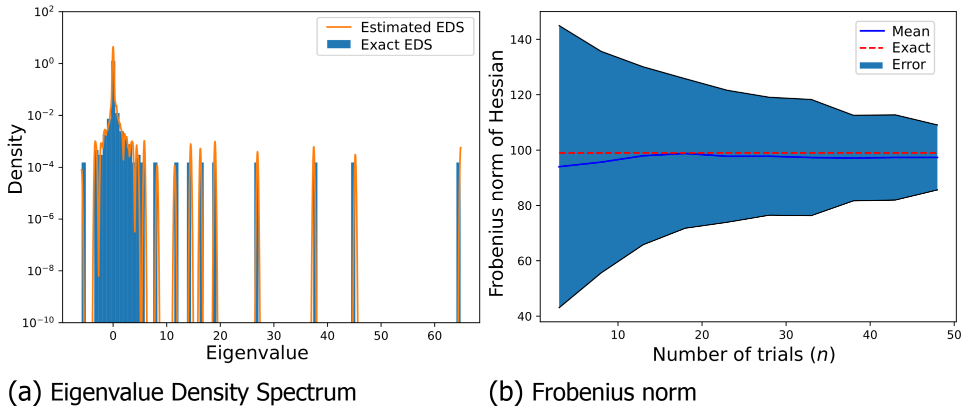
where, $\mathbf{J}_k$ is the lower triangular matrix obtain from the Cholesky factorization of the matrix $\mathbf{S}_k^T \mathbf{B}_0 \mathbf{S}_k + \mathbf{L}_k \mathbf{D}_k^{-1} \mathbf{L}_k^T$. Rearranging Equation B.1 so as to include the factorization in Equation B.6, we get the final formula implemented in the actual code of L-BFGS-B.

$$\mathbf{B}_k = \delta_k \mathbf{I} - \begin{bmatrix} \mathbf{Y}_k & \delta_k \mathbf{S}_k \end{bmatrix} \begin{bmatrix} -\mathbf{D}_k^{0.5} & \mathbf{D}_k^{-0.5} \mathbf{L}_k^T \\ 0 & \mathbf{J}_k^T \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{D}_k^{0.5} & 0 \\ -\mathbf{L}_k \mathbf{D}_k^{-0.5} & \mathbf{J}_k \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{Y}_k^T \\ \delta_k \mathbf{S}_k \end{bmatrix} \tag{B.7}$$

Using this form for the Hessian approximation is computationally efficient and also provides an easy oracle to computing the properties of the Hessian approximation made by the algorithm. This is because the Equation B.7 makes it easy to find the product of $\mathbf{B}$ with any vector, which is the matrix-vector product that we require. The relevant parts of L-BFGS-B that were necessary to compute the B-vector product (BVP) were implemented in Python. The correctness of the implementation was checked on a PixelModel where the $\mathbf{B}$ could be formed and inverted explicitly. The step direction as given by $\mathbf{B}^{-1}\vec{\nabla} f$ from our implementation was found to be in line with the actual step taken by the optimizer thereby giving credence to the correctness of the implementation.

B.5. Hyper-parameters for Hessian analysis

In order to set the hyper-parameters for these tools, testing was done on a smaller CNNModel without the dense layers and on a PixelModel, where the Hessian and its metrics could be calculated exactly, the representative results of which are shown in Figure B.2. Based on the tests, the parameters for the EDS were fixed as k=3, o=50 and σ^2 = 1e-4 (See section 2.4.2 for details about these hyper-parameters). For estimating the hessian metrics trace and norm, it was found that averaging over 3



(a) Eigenvalue Density Spectrum (b) Frobenius norm

Figure B.2: Testing the accuracy of the Hessian estimation tools. The test was conducted on a CNNModel with 8345 parameters. For estimating the EDS, the best hyper-parameters were k=3, o=50 and σ^2 = 1e-4. For estimating the Frobenius norm, the mean over 6 random seeds (as Mean) is shown alongside the confidence interval corresponding to 3 standard deviations (as Error).

random seeds for n = 50 trials was better (See section 2.4.2 for details about these hyper-parameters). From the paper by Ghorbani et al [46], it can be seen that the error between the EDS approximation and actual spectrum would decrease with an increase in both o as well as the number of parameters. Since the above tests worked for the PixelModel, it can be safely assumed that they will work better for the CNNModel which has significantly more parameters.

B.6. Test codes for appendix

In order to facilitate an easy representation of the results, each test case is assigned a unique code. The general form of this test code is "**XyZ**" where **X** denotes the model chosen, **y** is the value of the penalization factor and **Z** stands for the problem name. **X** can be either *NN* or *P*, denoting the CNNModel and PixelModel respectively. **y** can have values *1* or *3*, representing *p*=1 and *p*=3. Finally, **Z** can be either *A, B, D* or *E* corresponding to the 4 problems, as mentioned in Table 3.1. For example , the test code **NN3D** would represent the case of the TO problem D (Causeway bridge) with *p*=1 tested on the CNNModel.

B.7. Hyper-parameter optimization

In order to fix the step sizes for both SGD and Hessian descent, it was necessary to perform a rough hyper-parameter optimization. This was achieved using the *Hyperopt* [70] module of Python. Within the scope of this module, the main choices to be made were the search space, the algorithm to perform hyper-parameter optimization and number of trials. The search space was specified to be in the range of $[-6, 0.3]$ for CNNModels and $[-3, 1]$ for PixelModels in steps of 0.1 and was spec-

ified using the *quniform* function. The step size was taken as 10^x where x is the value chosen from the search space. This range was set based on previous manual tests done on both the CNNModel and PixelModel to approximately estimate their working ranges. The algorithm chosen was called *tpe* which stands for Tree of Parzen Estimators. The number of trials was set to 30. The objective for the optimization was chosen as the median final loss over some random seeds obtained by running an optimizer for a particular number of iterations. The detailed information regarding this is available in Table B.1, Table B.2 and Table B.3.

B

Test code	Max_iterations	Seeds	Best step size
NN3A	500	5	2.5e-4
NN3B	500	5	4.3e-3
NN3D	400	3	1e-3
NN3E	400	3	1.3e-3
NN1A	500	3	0.252
NN1B	500	3	1.585
NN1D	200	3	1.995
NN1E	200	3	0.5
P3A	500	5	0.1
P3B	500	5	1.19
P3D	200	3	1
P3E	200	3	1
P1A	100	5	10
P1B	100	5	10
P1D	150	5	7.94
P1E	150	5	10

Table B.1: Best step sizes obtained from hyper-parameter optimization for SGD. \emph{Max_iterations} denotes the maximum number of optimization steps. Please refer section B.6 for information on the test codes.

B

Test code	Max_iterations	Seeds	Best step size
NN3A	500	5	0.004
NN3B	500	5	0.016
NN3D	200	3	0.01
NN3E	200	3	0.016
NN1A	200	3	0.2
NN1B	200	3	0.5
NN1D	200	3	0.63
NN1E	200	3	0.63
P3A	500	5	0.126
P3B	500	5	0.398
P3D	200	3	0.32
P3E	200	3	0.4
P1A	200	3	1
P1B	200	3	0.63
P1D	200	3	1
P1E	200	3	1.26

Table B.2: Best step sizes obtained from hyper-parameter optimization for Hessian descent with m=1 (*HD_M1*). \emph{Max_iterations} denotes the maximum number of optimization steps. Please refer section B.6 for information on the test codes.

Test code	Max_iterations	Seeds	Best step size
NN3A	500	6	0.005
NN3B	500	6	0.0063
NN3D	200	3	0.025
NN3E	200	3	0.02
NN1A	300	6	0.25
NN1B	300	6	0.32
NN1D	200	3	0.32
NN1E	200	3	0.4
P3A	500	6	0.1
P3B	500	6	0.25
P3D	200	3	0.159
P3E	200	3	0.032
P1A	300	6	1
P1B	300	6	0.79
P1D	200	3	1
P1E	200	3	0.79

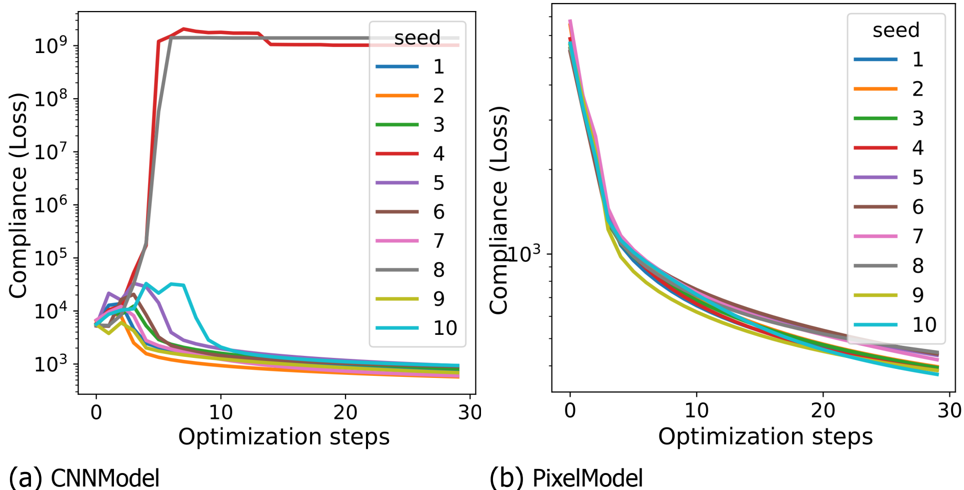
Table B.3: Best step sizes obtained from hyper-parameter optimization for Hessian descent with m=10 (*HD_M10*). *Max_iterations* denotes the maximum number of optimization steps. Please refer section B.6 for information on the test codes.

C

Supplementary information for chapter-4

This appendix will include the additional results to the ones presented in chapter 3.

C.1. Loss curves



(a) CNNModel (b) PixelModel

Figure C.1: SGD may diverge for some seeds for the CNNModel and not for the PixelModel. The results are from problem D with p=3.

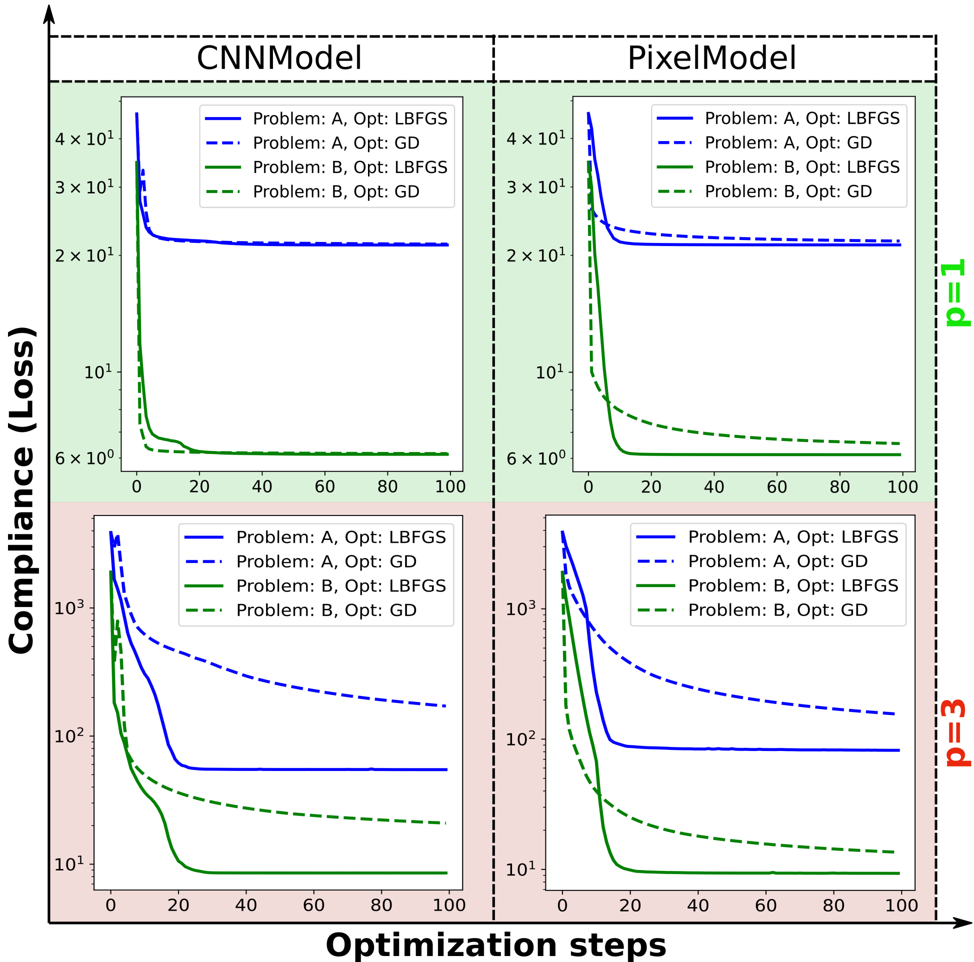


Figure C.2: Loss curves comparing PixelModel and CNNModel when the smaller problems A and B were optimized using L-BFGS and GD. The top row corresponds to the case with penalization = 1 and the bottom row is for p=3. The dashed lines represent GD while solid lines are for L-BFGS.

C.2. Eigenvalue Density Spectrum

This section will compare the EDS of the actual Hessian with that of the approximation made by L-BFGS with m=1 (*LBFGS_M1*). All these plots were estimated using Stochastic Lanczos Quadrature and matrix-vector product oracles.

C.3. Linear interpolation

More results obtained from applying linear interpolation between two iterates are provided in this section for select test cases.

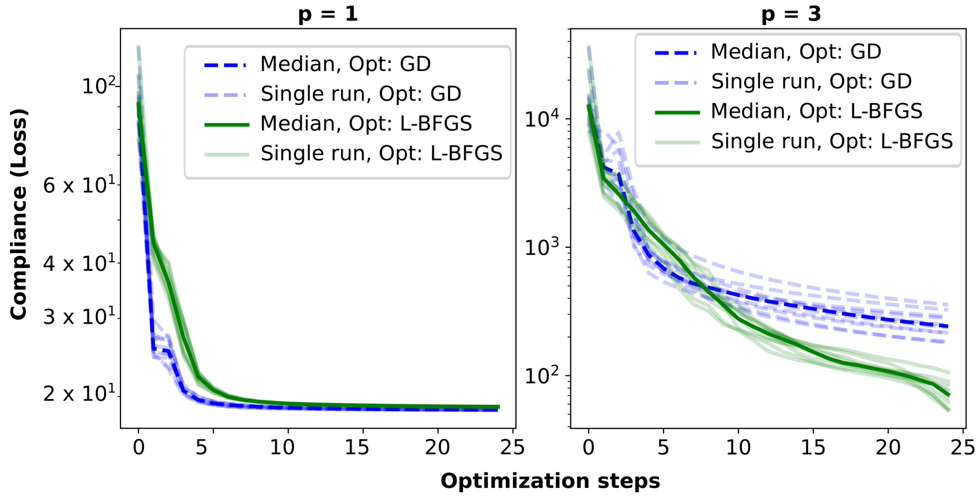


Figure C.3: Plot showing the loss curves for the individual runs for 10 random seeds. The results are for problem D on CNNModel. The left plot shows the case for a convex physics part where each run converges to the same loss value.

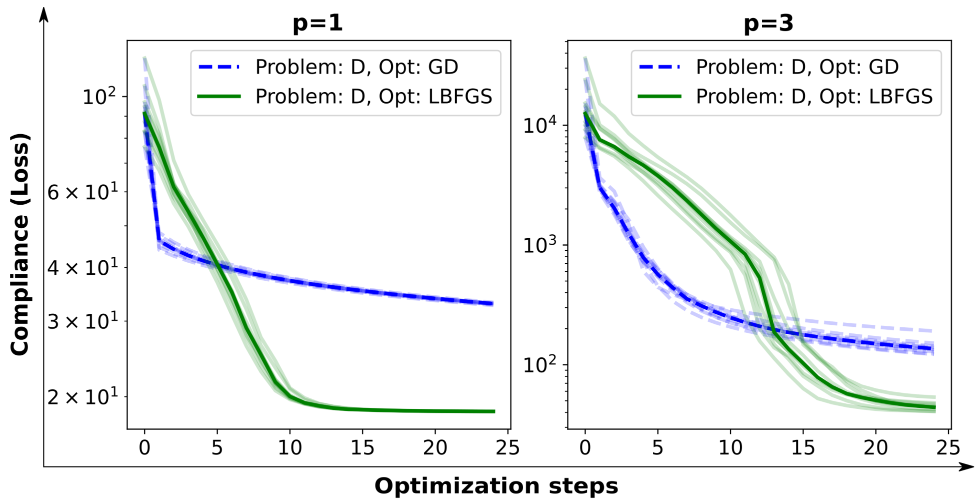


Figure C.4: Plot showing the loss curves for the individual runs for 10 random seeds. The results are for problem D on PixelModel. The left plot shows the case for a convex physics part where each run converges to the same loss value.

C.4. 2D projections

Results from projecting the loss landscape near the initialization onto random directions is provided in this section for select test cases..

C

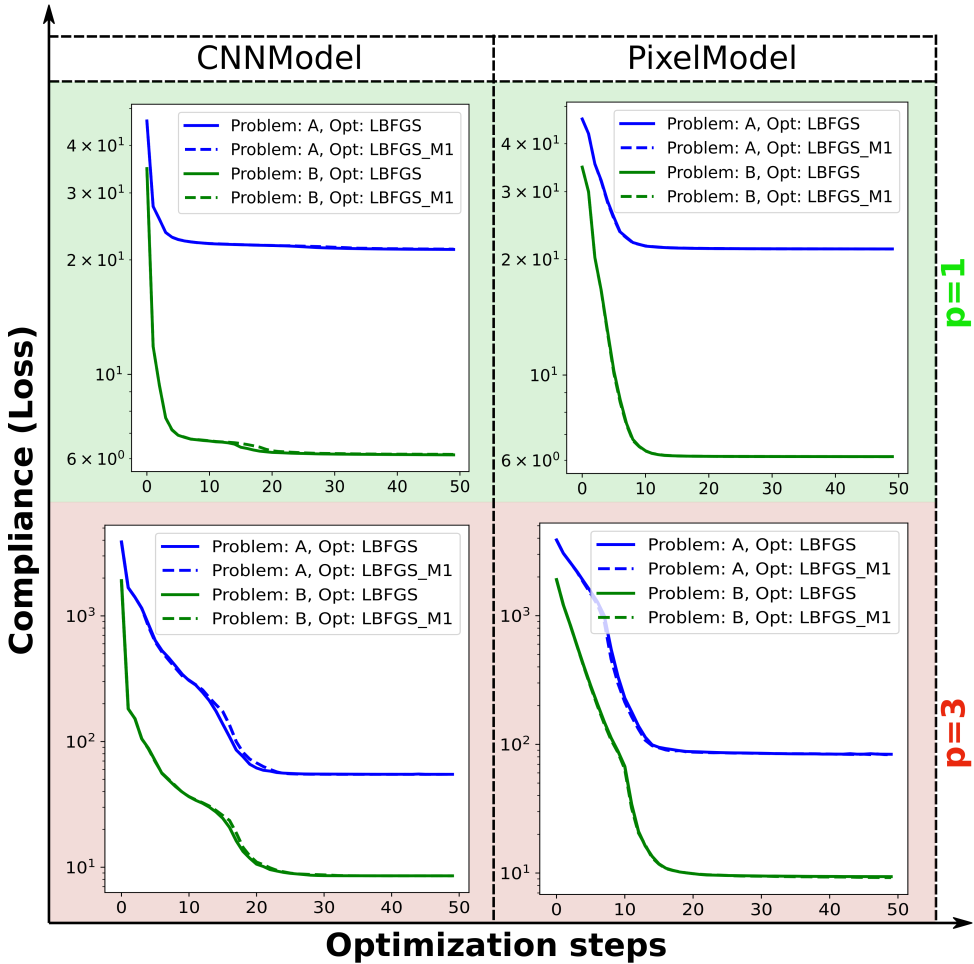


Figure C.5: Loss curves comparing the cases when the hyper-parameter of L-BFGS is varied when tested on the small problems. *LBFGS_M1* denotes the case with $m = 1$ while *LBFGS* is the default case with $m = 10$.

C.5. Distance and gradient norm

The distance travelled by the optimizers from initialization and the associated gradient norm values at each iterate are provided in this section for select test cases.

Figure C.6: Loss curves for Hessian Descent (*HD*) and Gradient Descent with line search (*GD_LS*) for the small problems. The subtexts of *M1* and *M10* correspond to the cases when the hyper-parameter m is set to 1 and 10 respectively.

C

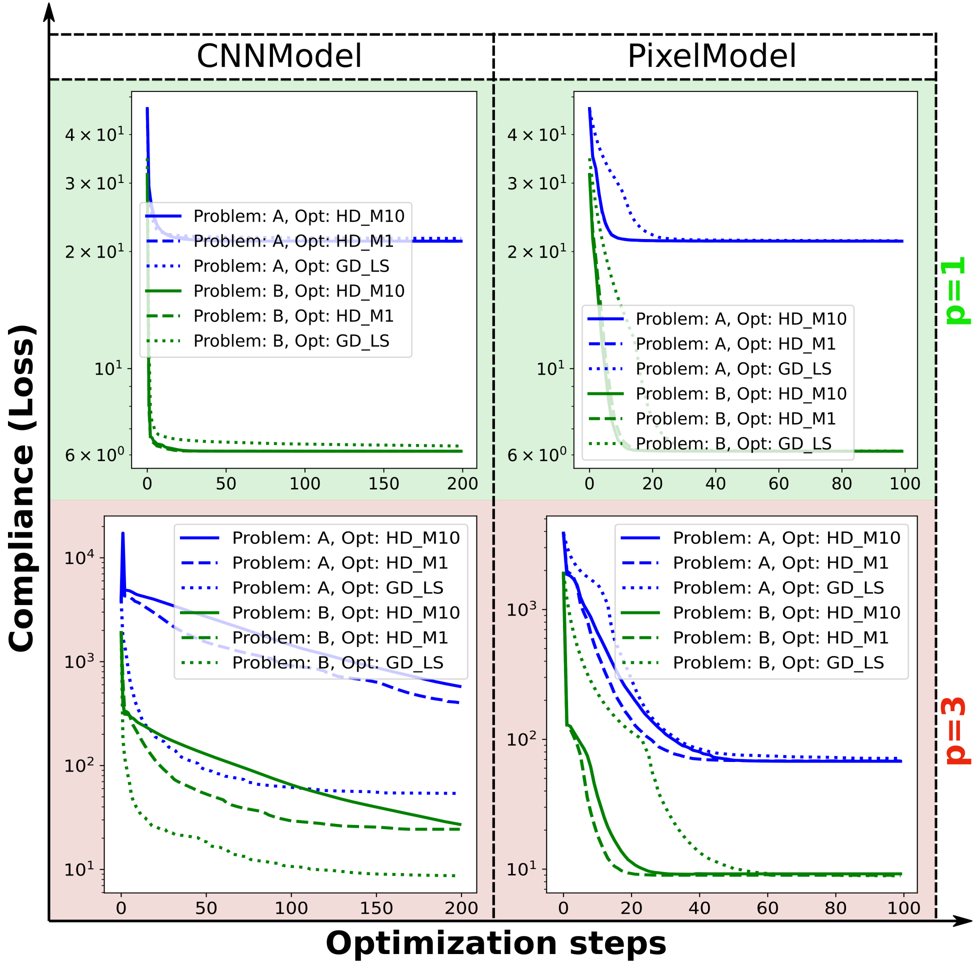


Figure C.7: Loss curves for Hessian Descent (*HD*) and Gradient Descent with line search (*GD_LS*) for the small problems. The subtexts of *M1* and *M10* correspond to the cases when the hyper-parameter m is set to 1 and 10 respectively. The plots have been zoomed in to clearly indicate the convergence of the different optimizers.

C

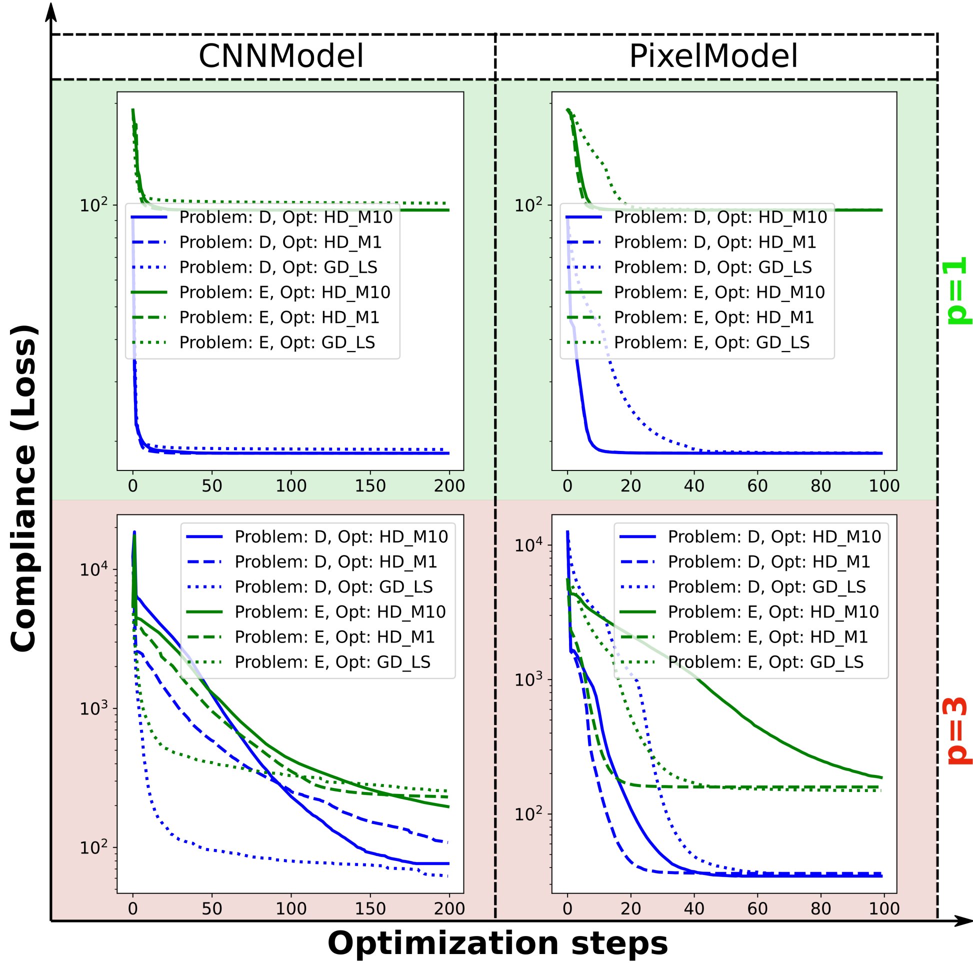


Figure C.8: Loss curves for Hessian Descent (*HD*) and Gradient Descent with line search (*GD_LS*) for the large problems. The subtexts of *M1* and *M10* correspond to the cases when the hyper-parameter m is set to 1 and 10 respectively. The plots have been zoomed in to clearly indicate the convergence of the different optimizers.

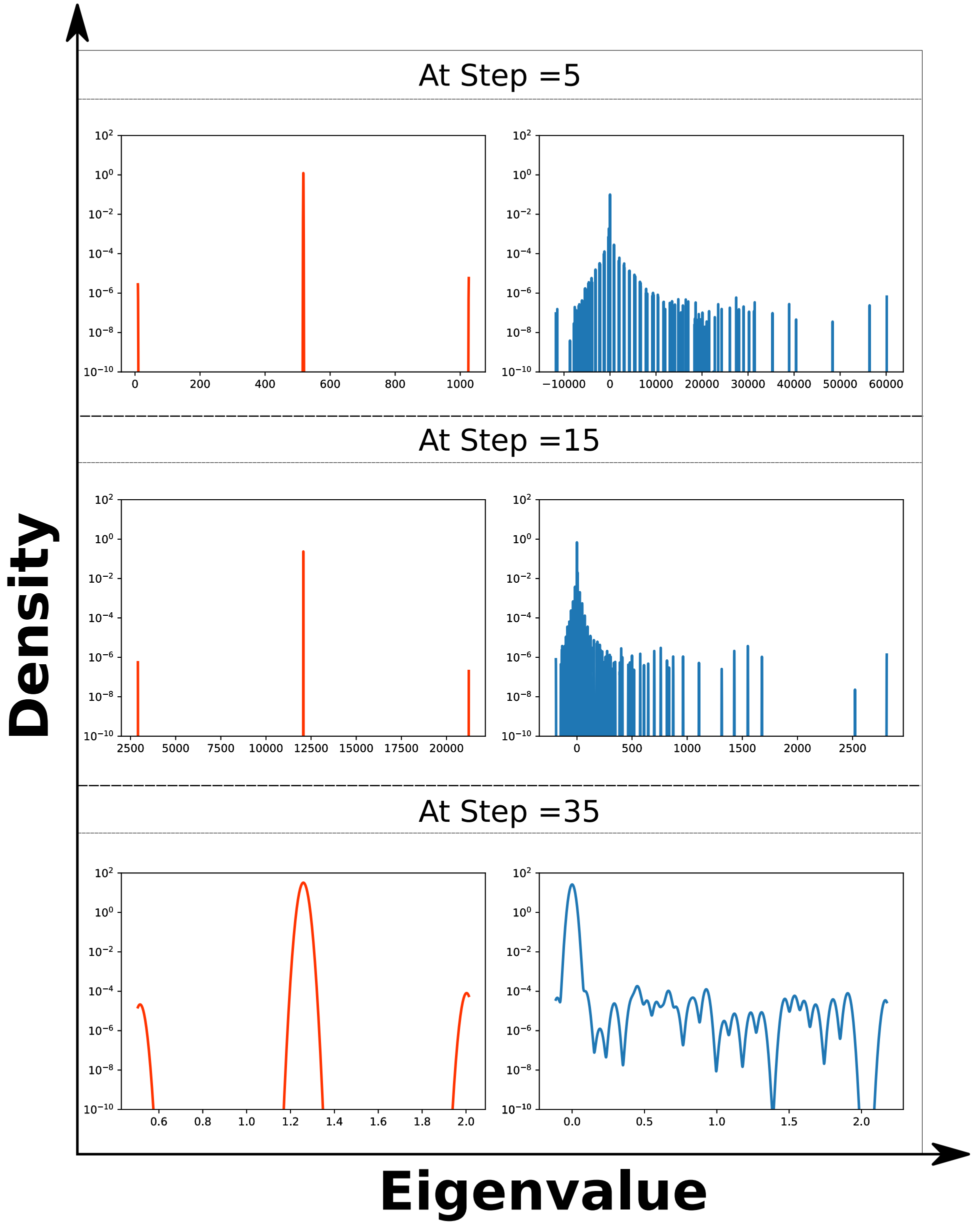


Figure C.9: Comparing the EDS of the actual Hessian (blue-right) with the L-BFGS approximation (red-left). This plot is for problem A, CNNModel, *p*=3 and run using *LBFGS_M1* (Test code: NN3A).

C

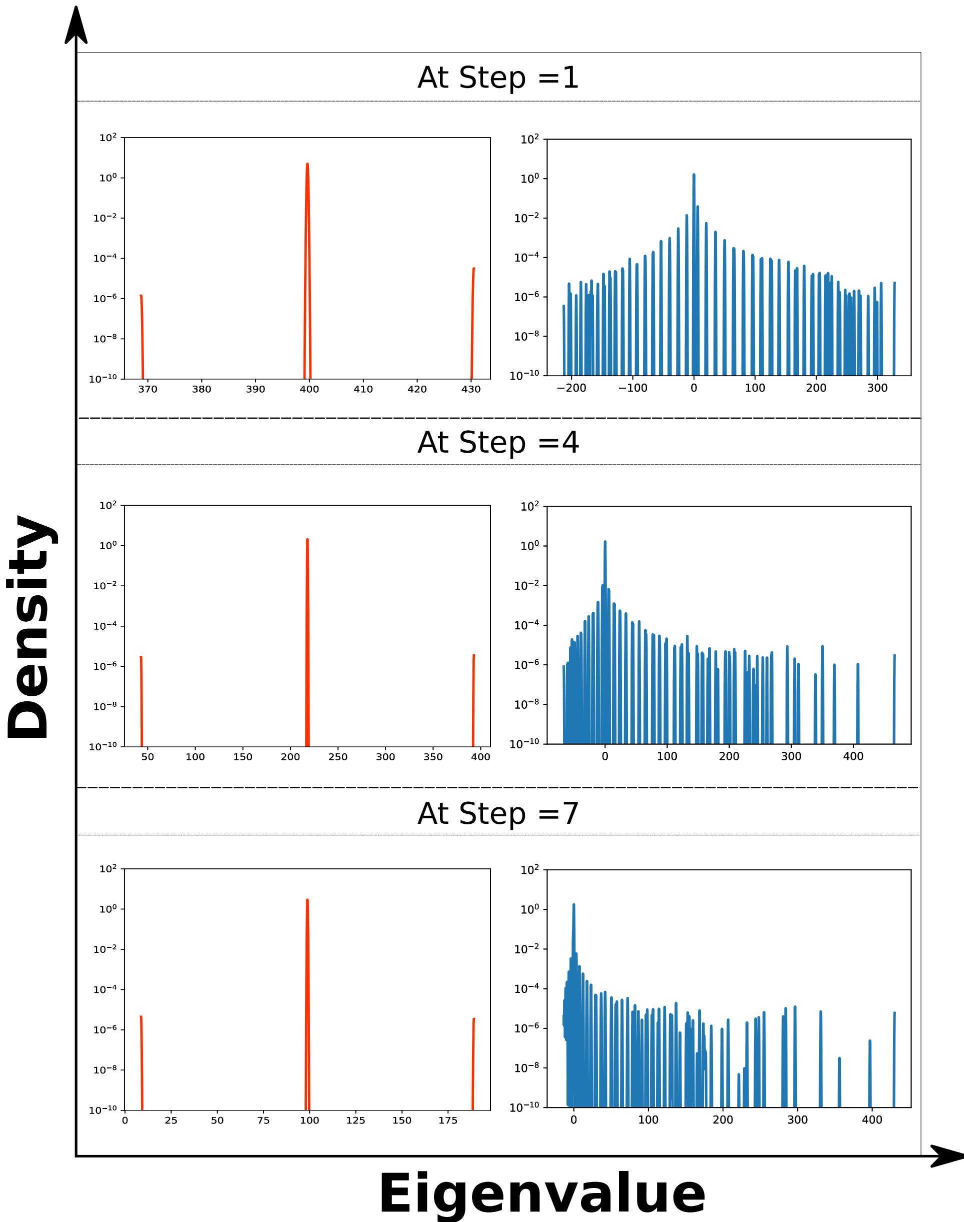


Figure C.10: Comparing the EDS of the actual Hessian (blue-right) with the L-BFGS approximation (red-left). This plot is for problem A, CNNModel, p=1 and run using *LBFGS_M1* (Test code: NN1A).

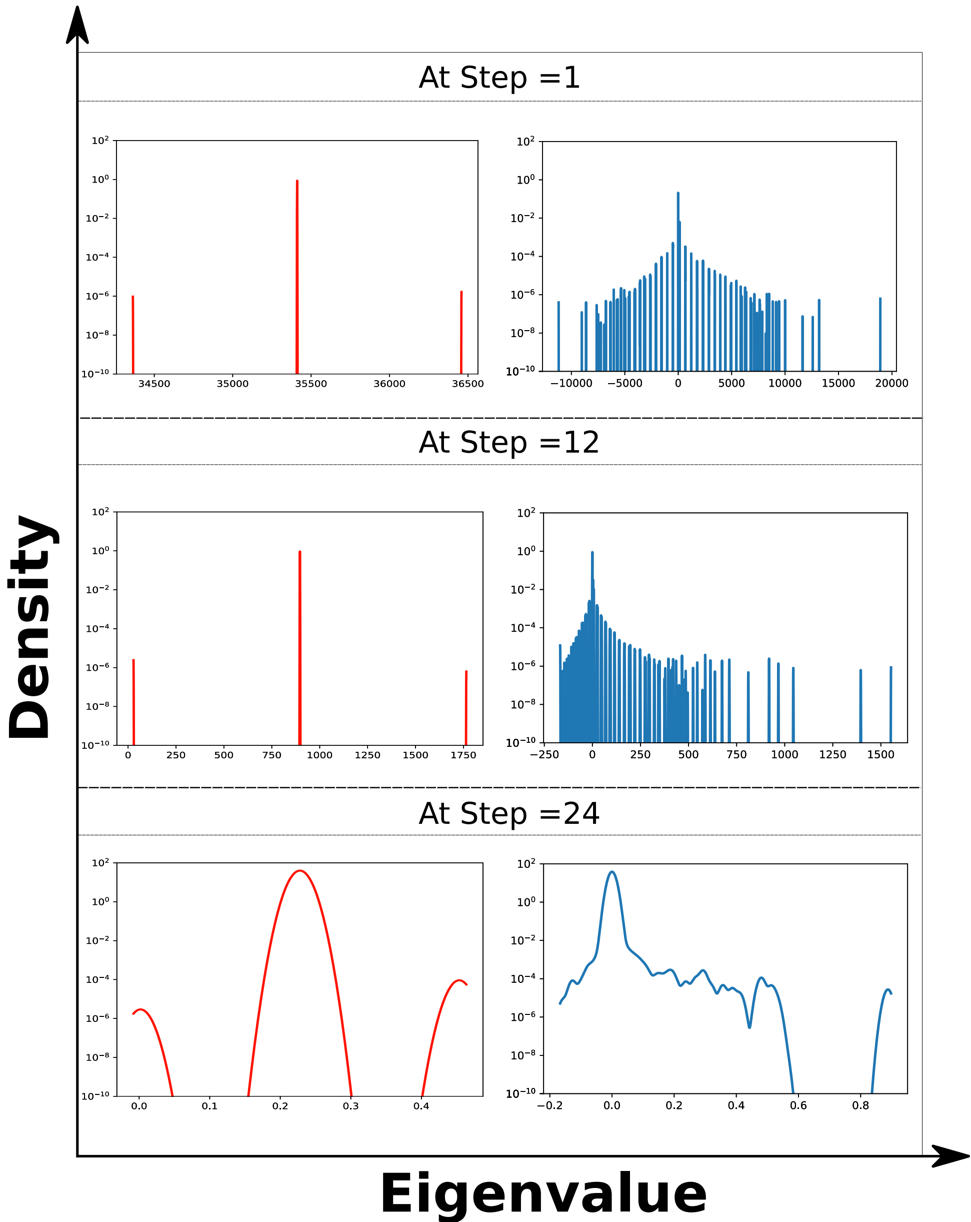


Figure C.11: Comparing the EDS of the actual Hessian (blue-right) with the L-BFGS approximation (red-left). This plot is for problem B, PixelModel, p=3 and run using *LBFGS_M1* (Test code: P3B).

C

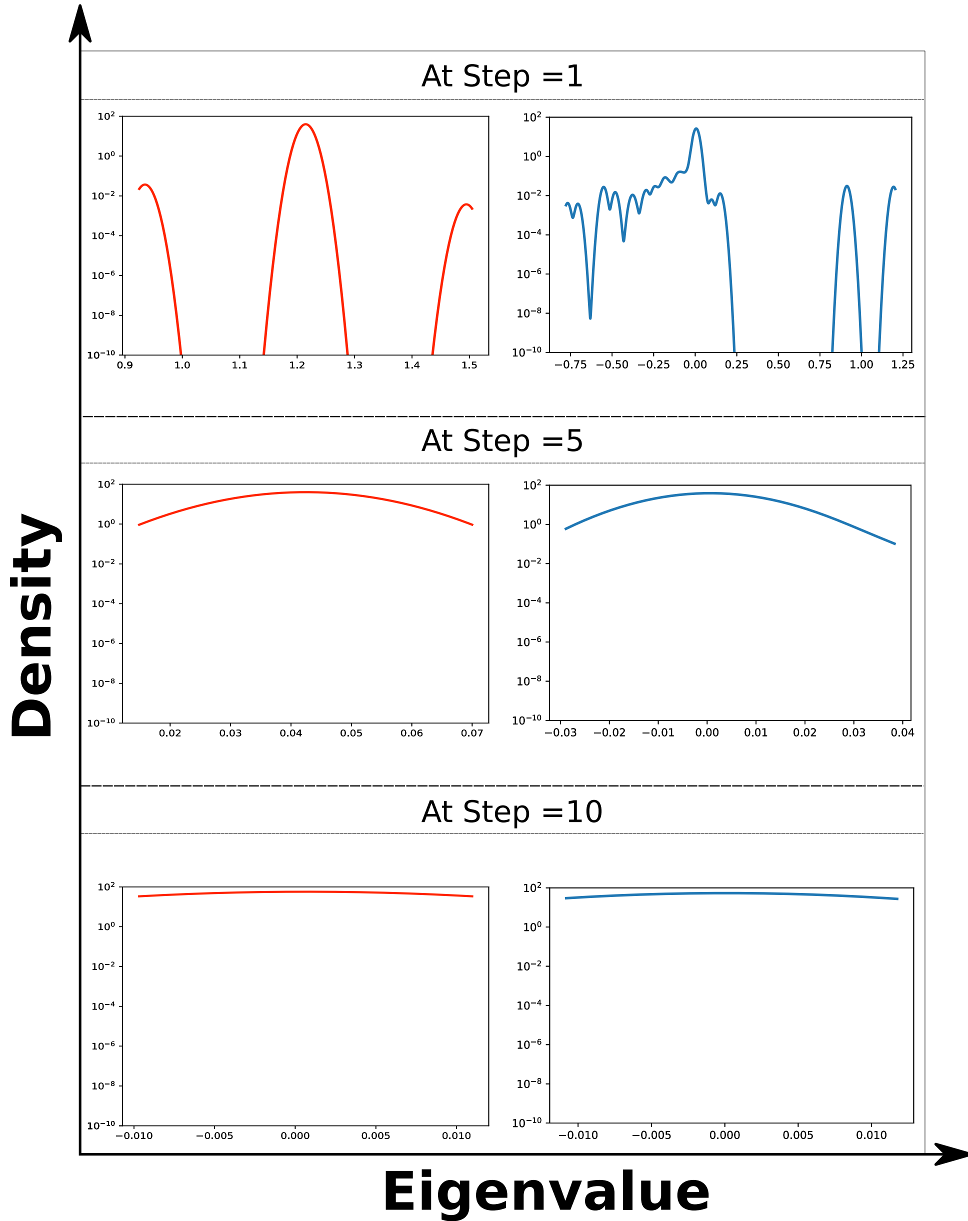


Figure C.12: Comparing the EDS of the actual Hessian (blue-right) with the L-BFGS approximation (red-left). This plot is for problem B, PixelModel, p=1 and run using *LBFGS_M1* (Test code: P1B). The eigenvalues become close to zero as early as step 10. The Hessian approximation is always positive definite but the small negative eigenvalues at step=10 are an artefact of the plotting process.

C

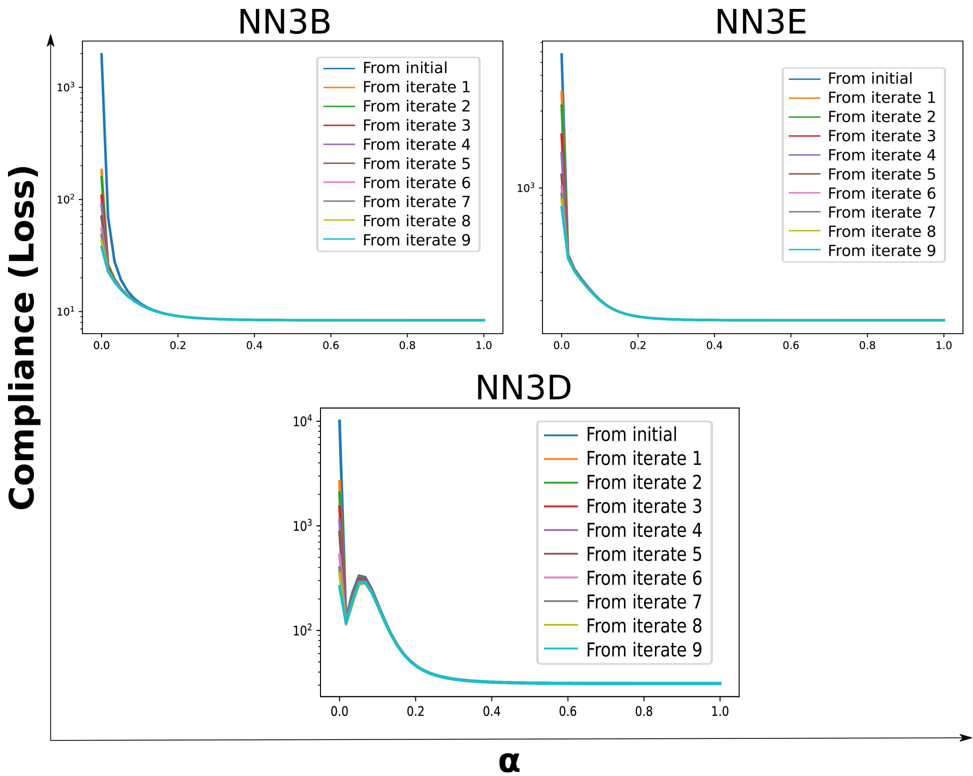


Figure C.13: Linear interpolation from the optimizer's iterates to the obtained minima for CNNModel and $p = 3$ case. Notes on the test codes used are available in section B.6. The only exception to a convex subspace was the case for NN3D, where a clear bump is visible.

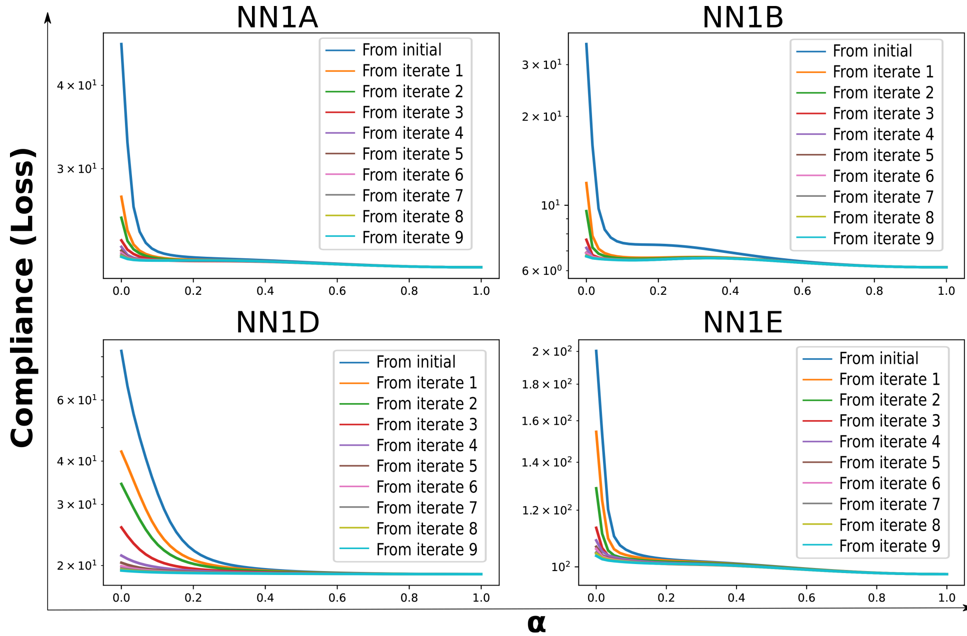


Figure C.14: Linear interpolation from the optimizer's iterates to the obtained minima for CNNModel and $p = 1$ case for a particular run. Notes on the test codes used are available in section B.6.

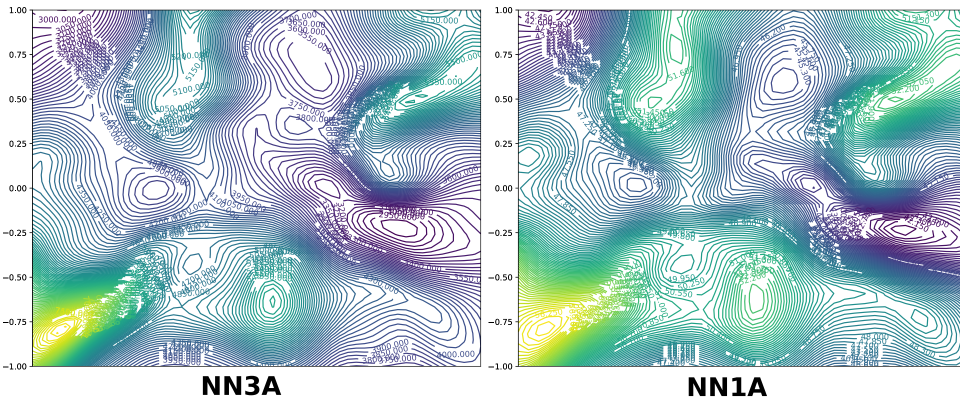
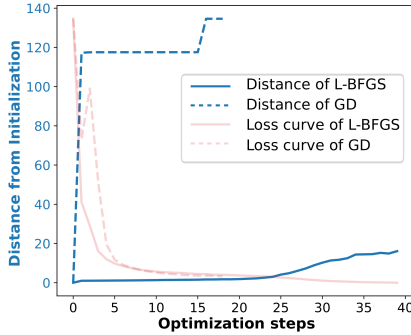
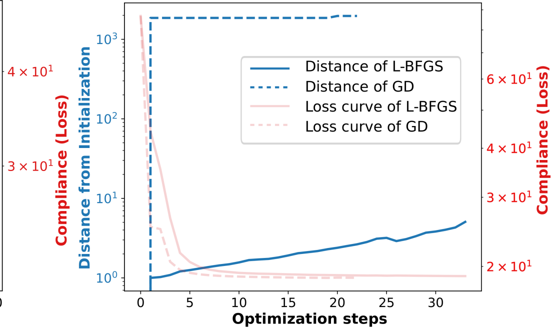


Figure C.15: 2D projection of the landscape around the initialization onto filter-normalized random vectors [42] for CNNModel. Notes on the test codes used are available in section B.6.

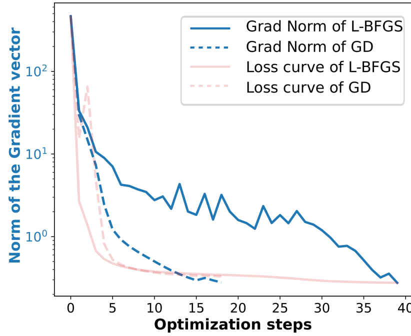
C



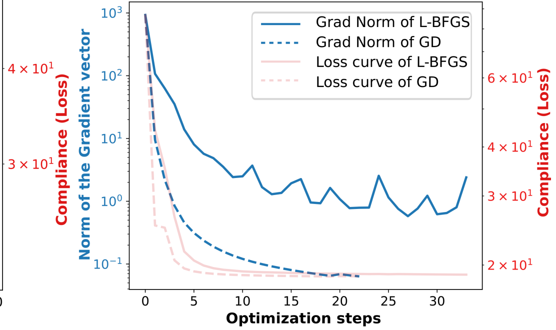
(a) Problem A, p=1



(b) Problem D, p=1



(c) Problem A, p=1



(d) Problem D, p=1

Figure C.16: Distance and the *L2* norm of the gradient (median over 3 seeds) moved by L-BFGS and GD from the initialization point for the CNNModel. The loss curves (median over 10 seeds, in red) is shown in reduced opaqueness to show the correlations between the reduction in loss, gradient norm values and the distance moved. Figure C.16b is shown in log scale because GD moved far too much as compared to L-BFGS. Results are shown for problems A and D for $p = 1$.