Spring 1-1-2019

Standard and Inception-Based Encoder-Decoder Neural Networks for Predicting the Solution Convergence of Design Optimization Algorithms

Nathanial James O'Neill
University of Colorado at Boulder, naon3943@colorado.edu

Standard and Inception-based Encoder-Decoder Neural Networks for Predicting the Solution Convergence of Design Optimization Algorithms

by

Nathanial James O'Neill

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Aerospace Engineering Sciences

2019

This thesis entitled:
Standard and Inception-based Encoder-Decoder Neural Networks for Predicting the Solution Convergence of Design Optimization Algorithms
written by Nathanial James O'Neill
has been approved for the Department of Aerospace Engineering Sciences

Prof. Kurt Maute

Prof. Alireza Doostan

Prof. John Evans

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

O'Neill, Nathanial James (M.S., Aerospace Engineering Sciences)

Standard and Inception-based Encoder-Decoder Neural Networks for Predicting the Solution Convergence of Design Optimization Algorithms

Thesis directed by Prof. Kurt Maute

The goal of this work is to investigate the ways in which the capabilities of machine learning algorithms, specifically those of neural networks, can be leveraged to enhance the performance of design optimization algorithms – specifically those of topology optimization.

A recent boom of interest in design optimization has occurred, coinciding with the arrival and development of advanced manufacturing techniques (such as 3D printing and additive manufacturing) which are compatible with the designs generated by these algorithms. Neural networks have seen an even larger boom in interest and development for their ability to act as "universal function generators;" in other words, for their ability to learn highly non-linear functions that approximate the behavior of extremely complex systems. Merging design optimization algorithms with the capabilities of neural networks poses several distinct possibilities: drastically reducing optimization time by predicting solution convergence; up-scaling solution resolution using Generative Adversarial Networks (GAN's); predicting solutions with no iteration; predicting and recognizing features in the optimized solution, just to name a few.

In this thesis, three neural network architectures are tested for their ability to act as solution convergence predictors of a density-based topology optimization solver. The problem is posed as an image segmentation problem, and the neural networks are all trained on a 40,000 example training set with each example containing 100 iterations from the open source optimization solver *Topy* (a data set created by Sosnovik *et al* [25]). The third network developed and tested is a novel hybrid network – an inception encoder-decoder network – which is found to outperform the other networks on the prediction task at hand.

Dedication

I would like to dedicate this thesis to all the individuals who helped and supported me – through the academics and through the life situations – to make this thesis possible.

Acknowledgements

I would like to acknowledge Prof. Kurt Maute for his support throughout this process. His knowledge and guidance were crucial to the formation and execution of this research. I would also like to thank Prof. John Evans for his guidance throughout the years. Having taken four of his classes, there is no professor responsible for more of my undergraduate and graduate work load, and no professor responsible for more of my learning and growth as a student and engineer.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

This chapter serves to provide the reader with an introduction to the major topics contained in this thesis, including a brief overview of topology optimization and machine learning, the motivation behind the research, the major results, and an overview of the structure of the contents.

1.1 Overview

1.1.1 Topology Optimization and Manufacturing

Human beings have long been fascinated, in areas ranging from art to engineering, by geometry and its relationship to function. There is something about the topic, on all levels, which is thoroughly captivating to our minds. Psychologically, a geometry with a specific function or meaning is a symbol. The cross of Christianity, the Christmas tree with a star on top, the swirling serpents of the yin-yang – all of these are demonstrations of function or meaning being ascribed to a geometry. This interest has likewise taken over our practice of engineering and design: each geometry we design looks the way it does so it can perform a certain function. Multiple geometries, however, are capable of performing a specific function, which then begs the question: how do we find an optimal geometry for a specific function or set of functions?

Motivated by this question, we can define *topology optimization*, generally speaking, as the numerical method for finding a geometry which optimally meets the constraints that specify its function. These constraints come in the form of maximum and minimum mass, displacements, temperatures, stresses, etc., as well as in manufacturability and manufacturing cost. While shape

optimization methods have been researched since the mid-1970's, topology optimization did not make its debut until 1988 when Bendsoe and Kikuchi published their work describing an approach for finding the structure with optimal material layout starting from a porous material distribution [8]. Quickly following, the SIMP (Solid Isotropic Material with Penalization) Method was developed as a means to alternatively solve the optimization problem as a *density* problem [6]. This method utilizes the elemental densities as the optimization variable, and the intermediate density values are heavily penalized to drive elemental density values to either 1 (full density) or 0 (no density).

Since the advent of design optimization algorithms, a fundamental problem has kept them from widespread adoption and implementation: manufacturing practices. Since the first day a human being used a tool to create something, we have been almost exclusively operating under a single manufacturing paradigm which goes something like the following: we create something by starting with more stuff than we need and then use a tool to remove what we don't. This manufacturing paradigm has allowed us to manufacture the world around us – but it is intensely limited in the designs it is capable of producing for the sole reason that we must be able to physically put a tool where we need to remove material. Certain methods of structural optimization stand compatible with this manufacturing paradigm, but many of the incredible designs that modern methods of design optimization are capable of creating stand far out of reach.

With the advent and development of 3D printing, additive manufacturing, and other advanced manufacturing techniques, humanity has leapt into a new era of how we turn our ideas and designs into reality. Manufacturing techniques are no longer constrained by the need to machine material with a tool, meaning engineers are now able to manufacture the previously non-manufacturable designs of optimization algorithms.

1.1.2 Machine Learning and Neural Networks

What is machine learning, on the highest level? In 1959, Arthur Samuel, who created the first computer learning program to play checkers, provided the following definition:"[Machine learning is] the field of study that gives computers the ability to learn without being explicitly programmed."

A better – or at least more specific – definition was offered by Tom Mitchell, who said, "A computer is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." Machine learning, then, is the subset of the larger field of artificial intelligence which encompasses the algorithms which, when employed, provide computers the ability to "learn."

A few of the most commonly used machine learning algorithms are:

(1) k-Nearest Neighbors

(2) Support Vector Machines (SVM's)

(3) Naive Bayes Classifiers

(4) Decision Trees

(5) Artificial Neural Networks (ANN's)

Note that this thesis will focus exclusively on the algorithms and architectures which make up Artificial Neural Networks.

Machine Learning algorithms can generally be divided into two main categories: supervised learning and unsupervised learning. Supervised learning algorithms are provided an "answer key," so-to-speak, from which they are able to learn the relationships within the provided data set. Unsupervised learning algorithms, conversely, are deployed in data sets to find certain kinds of answers – they group like things together and find structure in data sets, for example. Supervised learning can then broken down into two further sub-categories: regression and classification. Regression problems seek to map input variables to some continuous function; in other words, regression problems are curve fitting problems. Classification problems seek to predict discrete outputs based on a given input; in other words, they try to map input variables into discrete categories. The most common unsupervised learning algorithms are:

(1) *Clustering Algorithms*: group subsets of a larger dataset by similarity based on certain variables.

(2) *Dimensionality Reduction*: project the given data into a space of fewer variables, which is a powerful tool for both computational and visualization reasons. Primary Dimensionality Reduction algorithm is PCA, or Principle Component Analysis.

(3) *Outlier Detection*: find data that doesn't fit the expected trend.

The work in this thesis draws exclusively from supervised regression and classification algorithms, which are the foundation of ANN's.

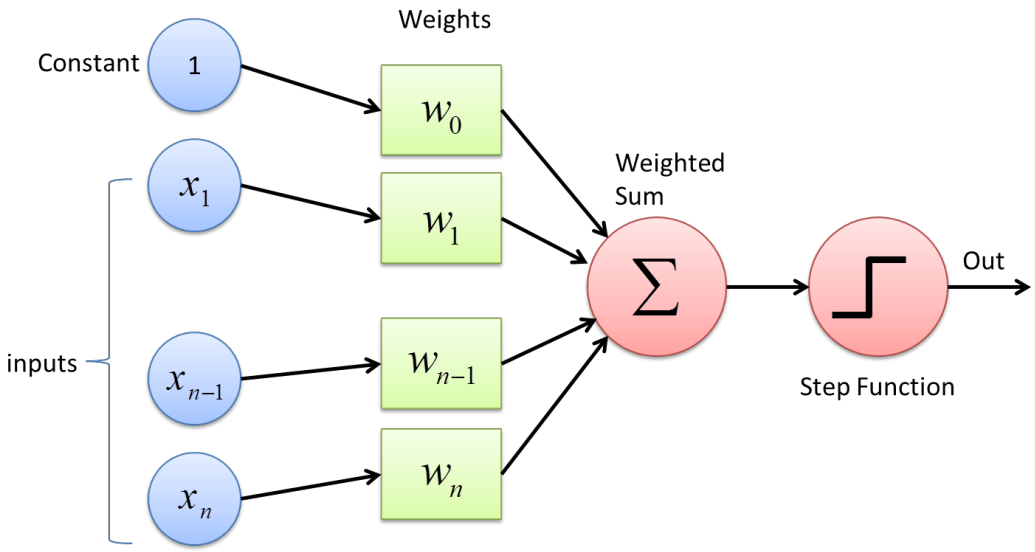

Figure 1.1: Perceptron Model, adapted from [22]

The idea of fitting a line to a data set seems, on the surface, to be a relatively trivial mathematical task. On a broader level, however, fitting a line to a curve is a specific instance of the ability to find a pattern, which is a core feature of systems that learn. Problems arise, mathematically, when the data set is sufficiently large and the curve we are trying to fit to it – the so-called "hypothesis" – becomes *highly* non-linear. Such problems, when solved with conventional algorithms, become enormously computationally expensive.

In the 1950's, researchers took note of something simple but significant: the human brain, and brains in general, are *exceptionally* good at finding patterns in highly complex data, which begged the question: what if the learning process of the human brain could be reconstructed

mathematically? This led to the so-called "Neuron Hypothesis" and the invention of the *perceptron* – the mathematical neuron – in 1958 by Frank Rosenblatt [21]. The perceptron, seen in Fig. 1.1, then evolved into the neuron, which is simply a perceptron with a continuous output, and was placed into networks with interconnecting lines by John Hopfield in 1982 – similar to how neurons in brains are connected.

Machine learning researchers were able to take these neural networks and improve them over the coming decades to the point where they could beat human beings at the most complicated games we have ever played. The earliest major acomplishment, arguably, was passed when IBM's Deep Blue beat Kasparov at chess in 1998. Google DeepMind's AlphaGo Master then beat Ke Jie, the No. 1 Go player in the world, at Go in 2017. A particularly impressive fact about the game Go, which yields testament to the level of intelligence required to play it well, is that there are significantly more possible moves in Go then there are atoms in the observable universe; in fact, there are approximately 10^{720} possible Go games for every atom in the observable universe [4].

These networks are so powerful they have permeated to nearly every aspect of our lives, from driving our cars and tractors to populating our Facebook news feeds with stories they predict we will find interesting.

Chapter 2 of this thesis will dive into some of the types, architectures, and training methods of Artificial Neural Networks.

1.2 Motivation: Leveraging Neural Networks for Design Optimization

The motivation behind investigating the applications on neural networks for design optimization algorithms is straightforward. Design Optimization algorithms can produce incredibly desirable results from an engineering perspective, and the field itself still contains an enormous amount of potential; however, the optimization problems these algorithms solve are highly non-convex by nature and generally run in enormous variables spaces, subsequently creating massively expensive problems to solve from a computational perspective. Neural networks, on the other hand, are essentially extraordinarily powerful non-linear function generators; or, in other words, they are

very good at fitting highly non-linear lines of best fit to any given data set. This poses a few interesting possibilities:

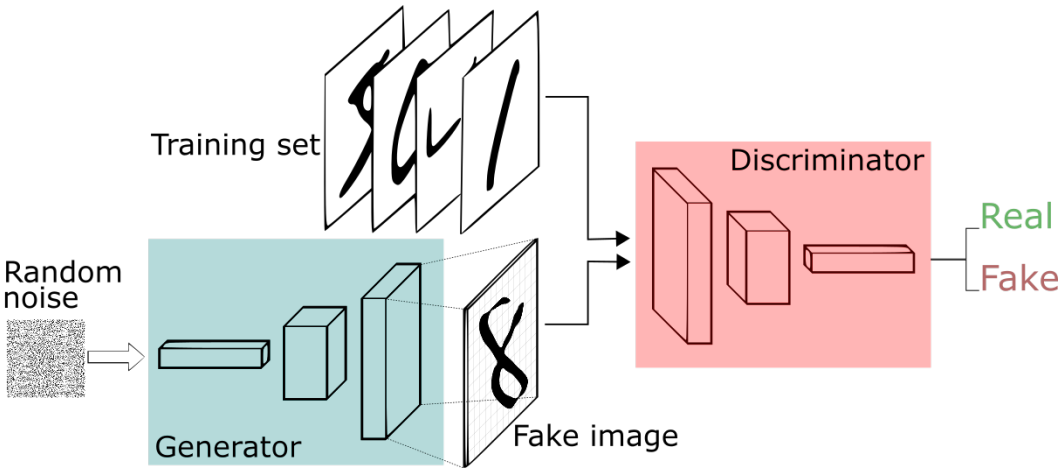

Figure 1.2: High-Level Daigram of a Generative Adversarial Network, adopted from [1]

(1) Predicting Solution with No Iteration: a neural network, provided an adequate training set, can learn the highly non-linear relationship between a design optimization problem (e.g. the shape, boundary, and loading conditions) and its topologically optimal solution. In a world where no computation limits existed and such a training set existed, this network could predict topologically optimal solutions to arbitrary design problems in a matter of seconds as opposed to days of high-performance computing time.

(2) Upscaling Solution Resolution: One of the largest factors in computation time of design optimization problems is the mesh refinement level of the finite element problem which runs in partnership with the optimization algorithm. The coarser the mesh, the faster the algorithm yields a a sufficiently optimal result. A certain kind of artificial neural network, a so-called Generative Adversarial Network (or GAN) seen in Figure 1.2, has the ability to learn how to generate sufficiently convincing outputs as to make them nearly identical to whatever "real" thing you are trying to get the network to learn to generate. The following example explains how such a network works: there is a "Discriminator Network" which is being trained to identify dog images from non-dog images and takes in two inputs, one

from a training set of labeled examples of dog images and the other from the "Generator Network," which is learning to generate realistic pictures of dogs. The Discriminator and Generator networks are then posed to play a "Min-Max" game with the cost function of the Discriminator: the Generator Network is trying to maximize the cost/loss of the Discriminator (by producing a realistic dog picture), and the Discriminator network is trying to minimize its own cost/loss (by becoming better at recognizing real dogs). The way the networks are posed puts them in competition, hence the "Adversarial" part of the name, and the trained Generator Network can produce incredibly accurate images (images with a high discriminator confusion rate). In the context of finite elements, such networks are capable of learning what a solution from a highly refined mesh looks like (mathematically speaking) and upscale the solution resolution of coarser meshes. Essentially, GAN's take a lower-resolution, computationally cheaper problem, and step up its final resolution, which could save greatly on computation time.

(3) Predicting Solution Convergence: While similar to (1), this method seeks to make large or small leaps through the solution space using the gradient of the solution field to inform its prediction. Instead of starting with the boundary and loading specifications of the design space, this problem setup lets the optimization algorithm provide it with a gradient field, and then predicts what the solution field will look like a certain number of iteration down the solution path. If the network is only predicting a few iterations ahead, the output of the network would pass directly back into the optimization algorithm, which would run several more iterations on the network's guess before handing it off to the network to make another incremental convergence prediction. Such a neural network could significantly decrease computation time while minimizing the amount of "uninformed guessing" the network in doing.

This thesis focuses exclusively on a neural network for (3) above, predicting solution convergence of design optimization algorithms.

1.3 Accomplishments

The research in this thesis has resulted in the following significant accomplishments:

(1) Three neural networks have been successfully trained to predict the solution convergence of density-based topology optimization solvers.

(2) The performance data of each network have been collected to provide an understanding of the general capabilities of such networks applied to the solution convergence problem.

(3) A novel convolutional neural network architecture – a hybrid inception encoder-decoder network – has been proposed, tested, and found to outperform standard encoder-decoder networks on the solution convergence problem.

1.4 Thesis Structure

Following this chapter, this thesis is laid out in the following manner:

(1) **Theoretical Background**: The mathematical theory behind density-based topology optimization is laid out, following by a higher-level description of the prerequisite machine learning theory.

(2) **Methodology**: A description of each experiment is laid out, including an overview of each network, its architecture, the data set used to train the network, and the training parameters.

(3) **Results and Discussion**: The results of each experiment are shown with several figures showcasing the prediction capabilities of the networks. The results are described, followed by a discussion of the implications of the results.

(4) **Conclusions and Future Work**: The work of the thesis is summarized, and the areas of interest for future work are laid out in the form of a list of research questions.

Chapter 2

Theoretical Background

2.1 Topology Optimization

The simplest forms of structural topology optimization come in the form of ground structure approaches for optimizing discrete truss structures. Such problems remove unnecessary bars from a "ground structure" until no more bars can be removed without violating the constraints imposed in the problem's formulation. This thesis will not cover ground structure approaches, and instead will focus on topology optimization of 2D *continuum* structures utilizing SIMP-based density methods.

The topology optimization problem we then seek to solve is as follows:

$$\min_{\mathbf{s}} \quad Z(\mathbf{u}, \mathbf{s}) \tag{2.1}$$

subject to:

$$g_k(\mathbf{u}, \mathbf{s}) \leq 0 \quad \text{for } k = 1, ...N_g \tag{2.2}$$

$$h_k(\mathbf{u}, \mathbf{s}) = 0 \quad \text{for } k = 1, ...N_h \tag{2.3}$$

where Z is the objective function, $\mathbf{u}$ is the state vector of the system, $\mathbf{s}$ is the vector of optimization variables, g_k are the inequality constraints of number N_g, and h_k are the equality constraints of number N_h. Note that there is an additionally bounding constraint on the optimization variables s_i such that:

$$s_i^{min} \leq s_i \leq s_i^{max}$$

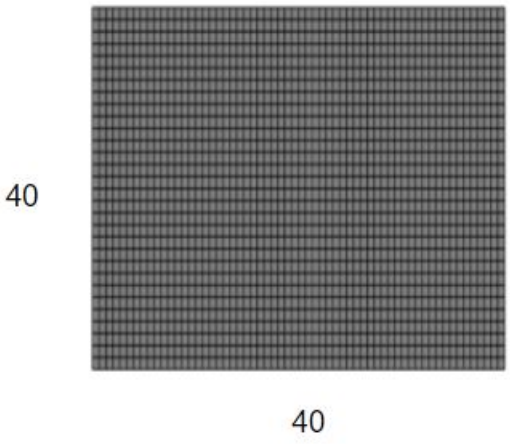

Figure 2.1: Design domain for generation of data set.

The SIMP method, which will be described in detail shortly, utilizes elemental densities as the optimization variable.

2.1.1 Geometry Description

In this thesis, the design domain of interest is a square continuum of isotropic, two-phase material (solid, void) discretized by a 40x40 finite element mesh of 2D, square elements, as seen in Figure 2.1.

2.1.2 SIMP Density Method

SIMP, which stands for **S**olid **I**sotropic **M**aterial with **P**enalization, is a density-based method for solving relaxed, continuous topology optimization problems. Two phases are available to describe the material layout within the design domain: void and solid material. The optimization variable **s** becomes the local element density $\bar{\rho}$, which is defined as continuous over the element:

$$\mathbf{s} = \bar{\rho}, \quad 0 < \bar{\rho} \leq 1$$

To navigate around the trivial solution of a uniform material distribution throughout the entire design domain, a volumetric or mass constrain is imposed upon the system. Assuming a mass constraint, the optimization problem then looks as follows:

$$\min_{\bar{\rho}} \; Z(\mathbf{u}, \bar{\rho}) \tag{2.4}$$

With mass constraint

$$h(\rho) = \int \rho \, dV - \bar{m} = 0 \tag{2.5}$$

Where $\bar{m}$ is a scalar value representing the maximum allowable mass of the system. The density filter, which imposes changes elemental densities, is applied through the local element stiffness matrix given by:

$$K_i^e = \tilde{\rho}_e^{\beta} K_0^e \tag{2.6}$$

Where K_0 is the initial elemental stiffness matrix and β is the penalization exponent (to be discussed in detail below). Also above, we see that $\bar{\rho}$ has turned suddenly into $\tilde{\rho}$. This $\tilde{\rho}$ represents the *linearly filtered* elemental densities, which provides the algorithm with higher numerical stability and prevents solution dependency on mesh refinement level. This filtered density $\tilde{\rho}_e$ is given by:

$$\tilde{\rho} = \frac{\sum w_{ij} \bar{\rho}_j}{\sum w_{ij}} \tag{2.7}$$

Here, the filter weight terms, w_{ij}, are defined as:

$$w_{ij} = r_f - |x_i - x_j| \tag{2.8}$$

Where $r_f = 1.6 \times h$ (h is the side length of the elements composing the mesh) is the filter radius, x_i is the location of the current element, and x_j is the location of the adjacent element. The factor of 1.6 is chosen by convention.

The problem is initialized with intermediate densities and the density variables are then iteratively updated and penalized for remaining intermediate using the density variable and the β exponent, both seen in (2.6). Figure 2.2 on the following page illustrates the relationship between normalized Young's Modulus and density over several values of β.

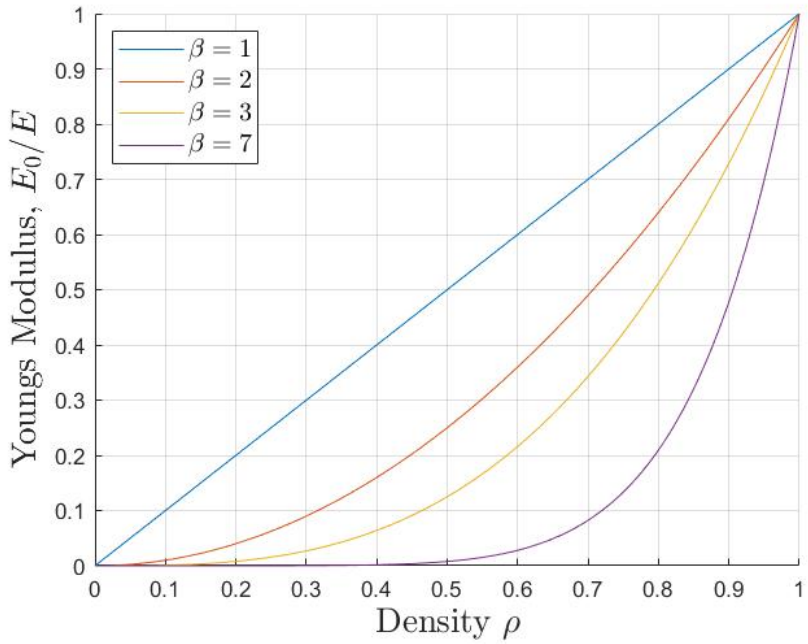

Figure 2.2: Beta power law material interpolation scheme

We see that as $\beta \to 1$, the resulting curve converges on the Hashin-Shtrikman bound. This figure begs the question, however: why not make β as large as possible? The answer lies in the value of the gradient of the Young's Modulus. As $\beta \to \infty$, the value of the gradient near $\rho = 0$ similarly becomes zero and the value of the gradient near $\rho = 1$ approach infinity. This phenomena leads to a non-convergent form of the material interpolation scheme. The optimal value, which provides both sufficient penalization without extreme gradients, is chosen by convention to be $\beta = 3.0$.

2.1.3 Optimization Algorithms

Generally speaking, there are two classes of optimization algorithms: gradient based and non-gradient based or gradient-free methods. Gradient-free methods tend to work well for topology optimization problems where the optimization variable may only take discrete values. Gradient-free methods include bound and branch methods, particle swarm methods, and evolutionary genetic algorithms. Although these methods work decently well for discrete problems with a small number of optimization variables, they tend to fall apart, so-to-speak, when there is a large number of opti-

mization variables (as there often is in topology optimization problems) and when the optimization variable is not bound to discrete values.

Gradient-based methods, as the name would imply, utilize the gradient of the system to find an optimal solution, and work well with both large numbers of optimization variables and non-discrete optimization variable values. Common gradient-based optimization algorithms include: gradient descent, conjugate gradient, non-linear conjugate gradient, BFGS, and others. Within gradient-based topology optimization, two optimization algorithms are commonly used, the latter of which being the more robust and widely used algorithm: the Optimality Critera Method (OCM) and the Globally Convergent Method of Moving Asymptotes (GCMMA). Generally speaking, these gradient-based optimization algorithms require the continuity and first-order differentiability of the objective and constraint functions.

2.1.3.1 Optimality Criteria Method (OCM)

The Optimality Criteria Method works well for density-based topology optimization problems, but tends only to converge when there is one constraint imposed on the system. A high level description of the Optimality Criteria method is given as follows.

The optimization problem is:

$$\min_{\mathbf{s}} \; Z(\mathbf{u}, \mathbf{s}) \tag{2.9}$$

With mass constraint

$$h(\mathbf{s}) = 0 \tag{2.10}$$

Where the optimization algorithm is:

0. Initialize algorithmic parameters (damping factor, q, and move limit, m).

1. Initialize primal variable, $\mathbf{s}^0$, such that $h(\mathbf{s}^0) = 0$.

2. Set iteration count, n, to zero: $n = 0$.

3. For $n > 0$, check convergence (i.e. check if $|\mathbf{s}^n - \mathbf{s}^{n-1}| < \varepsilon|\mathbf{s}^0|$).

4. Compute objective function and equality constraint partial derivatives, evaluated at the previous iteration value of $\mathbf{s}$, denoted s^n:

$$\left.\frac{\delta z}{\delta s_i}\right|_{s^n}, \quad \left.\frac{\delta h}{\delta s_i}\right|_{s^n} \tag{2.11}$$

5. Compute the Lagrange multiplier, η^n, using the bisection method such that $h(\tilde{\mathbf{s}}(\eta)) = 0$:

$$\tilde{\mathbf{s}}(\eta) = \begin{cases} max(s_i^L, s_i^n - m) & \mathbf{s}_i^n[B_i(\eta)]^q < max(s_i^L, s_i^n - m) \\ min(s_i^U, s_i^n + m) & s_i^n[B_i(\eta)]^q > min(s_i^L, s_i^n + m) \\ s_i^n B_i(\eta)^q & otherwise \end{cases}$$

where

$$B(\eta) = \frac{\left.-(\delta z/\delta s_i)\right|_{s^n}}{\left.\eta(\delta h/\delta s_i)\right|_{s^n}} \tag{2.12}$$

6. The new value of the primal variable is calculated based on the η from step 5:

$$s_i^{n+1} = s_i^n[B_i(\eta^*)]^q \tag{2.13}$$

7. Add one to iteration count ($n = n + 1$) and go-to step 3.

2.1.3.2 Dual Algorithm

Sequential programming optimization methods, such as GCMMA, split the optimization problem into sequential, separable convex approximations which are then solved using using a dual or primal-dual method. As such, it is necessary to give some background into dual algorithms along with GCMMA, which will be discussed in the following subsection. The general optimization

problem formulation can be given as follows: Find $\underline{s}$ such that:

$$\min_{\underline{s}} z(s) \tag{2.14}$$

Subject to:

$$g_j \leq 0 \quad j = 1, ..., N_j g \tag{2.15}$$

$$\underline{s} \in S = \{s_i \in \mathbb{R} \mid s_i^L \leq s_i \leq s_i^U,\ i = 1, ..., N_s\} \tag{2.16}$$

This formulation is then solved with a Dual Algorithm, where, with the utilization of a Lagrange function, the problem is formulated as:

$$\max_{\underline{\gamma}}(\min_{\underline{s}} L(\underline{s}, \gamma)) \tag{2.17}$$

Subject to:

$$\gamma_j \geq 0 \quad j = 1, ..., N_g \tag{2.18}$$

Given that the minimization problem is computational intensive, we employ an approximation at some $s^{(n)}$:

$$\min_{\underline{s}}\ L(\underline{s}, \gamma) = \phi(\underline{\gamma}) \approx \tilde{\phi}(\underline{\gamma}) = \min_{\underline{s}} \tilde{L}^{(n)}(\underline{s}, \underline{\gamma}) \tag{2.19}$$

where $\tilde{L}^{(n)}(\underline{s}, \underline{\gamma})$ is the local approximation such that:

- The problem formulation is globally convex (which guarantees a unique solution)
- The formulation is separable:

$$\tilde{L}^{(n)}(\underline{s}, \underline{\gamma}) = \sum_i \tilde{L}_i^{(n)}(\underline{s}_i, \underline{\gamma}) \tag{2.20}$$

- The formulation is analytic such that the minimization problem

$$\min_{s_i} \tilde{L}_i^{(n)}, \quad s_i^L \leq s_i \leq s_i^U \tag{2.21}$$

has a unique solution.

We have two possible approximations for the Lagrange function, L_i^a. Since the approximated Lagrange function should be convex, the approximation is selected based on the derivative of the objective and constraint with respect to the optimization variable: $\partial z/\partial s_i$ and $\partial g_j/\partial s_i$, respectively. Let $b = \{z, g_i\}$; then, the approximation scheme is given by:

$$\tilde{b}_j^{(n)} = b_j(s^{(n)}) + \begin{cases} (\partial b/\partial s_i)\Big|_{s^{(n)}}(s_i - s_n^{(n)}), & \text{if } (\partial b/\partial s_i) \geq 0 \\ -(s_i^{(n)})^2(\partial b/\partial s_i)\Big|_{s^{(n)}}((1/s_i) - (1/s_i^{(n)})), & \text{if } (\partial b/\partial s_i) < 0 \end{cases} \tag{2.22}$$

This is the so-called "hybrid-convex approximation," with which we approximate a parabola as a hyberbola at all points with a negative slope and as a line at all points with a positive slope. This approximation scheme guarantees convexity of the formulation. The approximated Lagrange function is then given by:

$$\tilde{L}_i^{(n)}(\underline{s}, \underline{\gamma}) = p_{zi}s_i + \frac{q_{zi}}{s_i} + \sum_j \gamma_j(p_{ji}s_i + \frac{q_{ji}}{s_i}) \tag{2.23}$$

And:

$$\tilde{\phi}^{(n)} = \sum_i \min \tilde{L}_i^{(n)} + \sum_j \gamma_j w_j \tag{2.24}$$

Where:

$$w_j = g_j(\underline{s}^{(n)}) - \sum_i p_{ji}s_i^{(n)} + \frac{q_{ji}}{s_i^{(n)}} \tag{2.25}$$

$$p_{ji} = \begin{cases} (\partial g_j/\partial s_i)\Big|_{\underline{s}^{(n)}}, & \text{if } (\partial g_{ji}/\partial s_i) < 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.26}$$

$$q_{ji} = \begin{cases} -(s_i^{(n)})^2(\partial g_j/\partial s_i)\Big|_{\underline{s}^{(n)}}, & \text{if } (\partial g_j/\partial s_i) < 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.27}$$

The objective coefficients p_{zi} and q_{zi} are produced similarly to those above.

Note that the solution of the minimization problem $\min_{s_i} \tilde{L}_i^{(n)}$ for a given $\underline{\gamma}$ is given analytically as:

$$s_i^* = \sqrt{\frac{q_{zi} + \sum_j \gamma_j q_{ji}}{p_{zi} + \sum_j \gamma_j p_{ji}}}, \quad \text{if } s_i^L \le s_i^* \le s_i^U \tag{2.28}$$

This provides us with three solution scenarios:

1. If both partials evaluated at the end points are negative, that is

$$(\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^L} < 0 \text{ and } (\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^U} < 0 \tag{2.29}$$

   (the approximated Lagrange function is monotonically decreasing) then $s_i^* = s_i^U$.

2. Conversely, if both partials evaluated at the end points are positive,

$$(\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^L} > 0 \text{ and } (\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^U} > 0 \tag{2.30}$$

   (the approximated Lagrange function is monotonically increasing)then $s_i^* = s_i^L$.

3. If the partials are different in sign such that

$$(\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^L} < 0 \text{ and } (\partial \tilde{L}_i^{(n)}/\partial s_i)\Big|_{s_i^U} > 0 \tag{2.31}$$

   (the approximated Lagrange function has a unique minimum in the boundary that is not the boundaries themselves) then the solution is given by Eqn. (20).

With $s_i^*(\gamma)$ known, we turn to evaluate:

$$\tilde{\phi}^{(n)}(\underline{\gamma}) = \underline{\gamma}^T \underline{w} + \sum_i \tilde{L}_i(s_i^*, \underline{\gamma}) \tag{2.32}$$

Which is solved numerically using a Non-Linear Programming (NLP) method.

2.1.3.3 Method of Moving Asymptotes (MMA)

Recall that our dual formulation leads to the following approximations for the objective and the constraint:

$$z^a = w_z + \sum_i \left(\frac{p_{zi}}{(u_i - s_i)} + \frac{q_{zi}}{(s_i - l_i)} \right) \tag{2.33}$$

$$g_j^a = w_j + \sum_i \left(\frac{p_{ji}}{(u_i - s_i)} + \frac{q_{ji}}{(s_i - l_i)} \right) \tag{2.34}$$

The approximated Lagrange function is built as the sum of the objective and constraint approximations:

$$L^a = z^a + \sum_j \gamma_j g_j^a \tag{2.35}$$

$$= (w_z + \sum_j \gamma_j w_j) + \sum_i \left(\frac{p_{zi} + \sum_j \gamma_j p_{ji}}{u_i - s_i} + \frac{q_{zi} + \sum_j \gamma_j q_{ji}}{s_i - l_i} \right) \tag{2.36}$$

$$= \tilde{w} + \sum_i L_i^a \tag{2.37}$$

Where:

$$w_z = z(\hat{s}) - \sum_i \left(\frac{p_{zi}}{u_i - \hat{s}_i} + \frac{q_{zi}}{\hat{s}_i - l_i} \right) \tag{2.38}$$

in which (2.39)

$$p_{zi} = \begin{cases} (\partial z / \partial s_i)\Big|_{\hat{s}_i} (u_i - \hat{s}_i)^2, & \text{if } (\partial z / \partial s_i) > 0 \\ 0, & otherwise \end{cases} \tag{2.40}$$

$$q_{zi} = \begin{cases} -(\partial z / \partial s_i)\Big|_{\hat{s}_i} (\hat{s}_i - l_i)^2, & \text{if } (\partial z / \partial s_i) < 0 \\ 0, & otherwise \end{cases} \tag{2.41}$$

The expression for w_j, p_{ji}, and q_{ji} are constructed similarly. From here, we can say:

$$L_i^a = \frac{\tilde{p}_i}{u_i - s_i} + \frac{\tilde{q}_i}{s_i - l_i} \tag{2.42}$$

Where:

$$\tilde{p}_i = p_{zi} + \sum_j \gamma_j p_{ji} \quad \text{and} \quad \tilde{q}_i = q_{zi} + \sum_j \gamma_j q_{ji} \tag{2.43}$$

The solution to the minimization problem, just as stated in the dual formulation, can be solved analytically. Setting the partial of the approximated Lagrange function with respect to the optimization variable equal to zero and solving for s_i, we find:

$$s_i = \frac{\tilde{p}_i l_i - \tilde{q}_i u_i \pm (u_i - p_i)\sqrt{\tilde{p}_i \tilde{q}_i}}{\tilde{p}_i - \tilde{q}_i} \tag{2.44}$$

Only the positive case need be considered implying:

$$s_i^* = s_i^+ = \frac{\tilde{p}_i l_i - \tilde{q}_i u_i + (u_i - p_i)\sqrt{\tilde{p}_i \tilde{q}_i}}{\tilde{p}_i - \tilde{q}_i} \tag{2.45}$$

Using the simplifications $\hat{p}_i = \sqrt{\tilde{p}_i}$ and $\hat{q}_i = \sqrt{\tilde{q}_i}$, this expression reduces to:

$$s_i^* = \frac{\hat{p}_i l_i + u_i \hat{q}}{\hat{p}_i + \hat{q}_i} \tag{2.46}$$

As stated in the dual algorithm formulation, we also check the signs of the derivative of the approximated Lagrange function at the upper and lower bounds to determine which of the three possible solutions is the solution for the given iteration.

Now, the further the upper and lower bound asymptotes are apart, the smaller the curvature of the approximated Lagrange function and the larger the search space. To control the overall convergence, the upper and lower bound asymptote locations are adjusted: the distance between the two is reduced if oscillations in convergence are observed, and the distance between the two is expanded for monotonic convergence. While $k = \{0, 1\}$:

$$l_i^{(k)} = s_i^{(k)} - \mu_a(s_i^U - s_i^L) \tag{2.47}$$

$$u_i^{(k)} = s_i^{(k)} + \mu_a(s_i^U - s_i^L) \tag{2.48}$$

Where $l_i^{(k)}$ and $u_i^{(k)}$ are the location of the lower and upper asymptotes for a given iteration k, respectively. If $(s_i^{(k)} - s_i^{(k-1)})(s_i^{(k-1)} - s_i^{(k-2)}) < 0$, then:

$$l_i^{(k)} = s_i^{(k)} - \mu_b(s_i^{(k-1)} - l_i^{(k-1)}) \tag{2.49}$$

$$u_i^{(k)} = s_i^{(k)} + \mu_b(u_i^{(k-1)} - s_i^{(k-1)}) \tag{2.50}$$

Otherwise:

$$l_i^{(k)} = s_i^{(k)} - \frac{1}{\mu_b}(s_i^{(k-1)} - l_i^{(k-1)}) \tag{2.51}$$

$$u_i^{(k)} = s_i^{(k)} + \frac{1}{\mu_b}(u_i^{(k-1)} - s_i^{(k-1)}) \tag{2.52}$$

2.1.4 Sensitivity Analysis

When using a gradient-based optimization algorithm, such as OCM or GCMMA, it is necessary to provide the optimization algorithm gradients of the objective and constraints with respect to the optimization variables. Methods for doing this include the finite difference method, the direct method, and the adjoint method. Due to the numerical inefficiency of the finite difference method, the adjoint method is the method of sensitivity analysis utilized in this thesis, and in most linearly elastic strain energy-based topology optimization problems.

Consider the following minimization problem:

$$\min_{\mathbf{s}} z(\mathbf{s}, \mathbf{u}) \tag{2.53}$$

Subject to:

$$g_j \leq 0 \quad j = 1, ..., N_j g \tag{2.54}$$

$$\mathbf{s} \in S = \{s_i \in \mathbb{R} \mid s_i^L \leq s_i \leq s_i^U,\ i = 1, ..., N_s\} \tag{2.55}$$

Differentiating the objective function $z(\mathbf{s}, \mathbf{u})$ utilizing the chain rule yields the following expression:

$$\frac{dz(\mathbf{s}, \mathbf{u})}{ds_i} = \frac{\partial z}{\partial s_i} + \frac{\partial z}{\partial \mathbf{u}} \frac{d\mathbf{u}}{ds_i} \tag{2.56}$$

The last term – the derivative of the state vector with respect to the optimization variable – is then solved by differentiating the residual of the governing equation, denoted by $\mathbf{R}$:

$$\frac{d\mathbf{R}(\mathbf{s}, \mathbf{u})}{ds_i} = \frac{\partial \mathbf{R}}{\partial s_i} + \frac{\partial \mathbf{R}}{\partial \mathbf{u}} \frac{d\mathbf{u}}{ds_i} = 0 \tag{2.57}$$

Where by definition:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}} = \mathbf{K} \tag{2.58}$$

Solving Equation (2.57) for $d\mathbf{u}/ds_i$ and plugging the results back into Equation (2.56), we arrive at the so-called adjoint solution:

$$\frac{dz(\mathbf{s}, \mathbf{u})}{ds_i} = \frac{\partial z}{\partial s_i} - \mathbf{a}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right) \tag{2.59}$$

Where $\mathbf{a}$ is found from solving the linear system referred to as the adjoint problem:

$$\mathbf{K}^T \mathbf{a} = \frac{\partial z}{\partial \mathbf{u}} \tag{2.60}$$

It is evident from examining the above equations that the adjoint problem need only be solved once for each objective and constraint function. For problems with a large number of optimization variables, the adjoint problem is the most computationally efficient problem to solve to compute the system sensitivities.

2.2 Artificial Neural Networks

As a term, machine learning encompasses a wide range of extraordinarily powerful and robust algorithms for establishing analytical models from given data sets. These models are then capable of making high-accuracy "decisions" – probabilistically speaking – with little or no need for human intervention. Artificial neural networks are but one algorithm encapsulated by the larger field of machine learning, but contain within them an enormous amount of potential as they are arguably the most powerful known algorithm for developing highly complex non-linear models. Motivated by the brain's profound ability for finding patterns and establishing models of its environment, neural networks are a mathematical representation of how the brain discover patterns and develop models.

The mathematical foundation of neural networks is found in a very simple mathematical problem: regression. Univariate examples of regression include problems such as:

(1) Housing Prices: find the relationship between house price and the square footage of the house.

(2) Profit Prediction: find the relationship between the profit of a business and the population size that the business serves.

Multivariate extensions of these examples would be:

(1) Housing Prices: find the relationship between house price and square footage, number of rooms, location, number of bathrooms, local school rating, etc.

(2) Profit Prediction: find the relationship between the profit of a business and the population size the business serves, the age of product consumers, the average income of product consumers, consumer demographics, etc.

Clearly the univariate problems are simple and do not provide extraordinarily useful information while the multivariate problems are significantly more complex and providing more useful

information. The regression problem here, in a simplistic sense, is what each neuron of a neural network is performing on a data set. Prior to diving into the theory of regression, consider the following list of notation and conventions:

- Let the term *training set* refer to the data of known inputs and outputs, or x, y pairs, from which the algorithm learns.
- Let the term *hypothesis* refer to the function, or prediction, computed by the algorithm which represents the learned relationship between the x's and $y's$ in the training set.
- Let $\hat{y}$ represent the hypothesis.
- Let m denote the number of training examples in the training set.
- Let X represent a vector containing the known x-values of the training set.
- Let Y represent a vector containing the known y-values of the training set.
- Let $(x^{(i)}, y^{(i)})$ represent the i^{th} training example in the training set.
- Let θ represent the vector of weights or model parameters that the algorithm is computing such that:

$$Y = \theta^T X \tag{2.61}$$

If the hypothesis is linear, linear algebra allows us to compute an analytical solution given by the Least Squares or Normal Equation:

$$\theta = (X^T X)^{-1} X^T Y \tag{2.62}$$

Should the problem be sufficiently large, such as when the parameter space is massive and the number of times the normal equation must be solved is also large, this method becomes computationally costly. Additionally, this equation only provides a linear hypothesis, and – generally speaking – most problems of interest in regression are not problems that require computing a linear

model. Where then do we proceed? The answer is simple: iterative optimization. Continuing our notation primer:

- Let $\mathcal{L}$ represent the *cost* or *loss* function of the optimization problem.

The optimization problem we then seek to solve can be defined as follows:

$$\min_{\theta} \ \mathcal{L}(\theta) \tag{2.63}$$

The primary loss function used in neural networks development or deep learning is the mean squared error function, which is given by the following:

$$\mathcal{L}(\theta) = \frac{1}{2m} \sum_{i}^{m} \left(\hat{y}(x^{(i)}) - y^{(i)}\right)^2 \tag{2.64}$$

The most widely used algorithm for solving this optimization problem is gradient descent and its variations: stochastic gradient descent, ADAM, etc. Gradient descent is a trivial optimization algorithm, and the details of these algorithms will not be discussed in this thesis. One problem that arises in any regression problem is the problem of underfitting and overfitting, or bias and variance as the two are referred to as in machine learning literature, respectively. The solution to preventing bias is to include more parameters in the model, and the solution to variance is referred to as *regularization*. Essentially, regularization methods add a term into the cost/loss function which increases (thus increasing the cost) as the magnitude of the weights learned by the algorithm increase, which corresponds with high variance. Several methods of regularization exist, one of the common being "L2 Regularization," which is seen as the last term in the following modified form of equation (2.64):

$$\mathcal{L}(\theta) = \frac{1}{2m} \sum_{i}^{m} \left(\hat{y}(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{m} ||W||_{\mathbf{2}}^2 \tag{2.65}$$

An important aspect of training these algorithms, which is the blanket term used to describe solving the optimization problem, is that the training set is typically broken down into multiple subsets:

(1) Training Set: A large portion of the original training set (60% or more) is used to train the algorithm.

(2) Development or Validation Set: A smaller chunk of the original training set (20% or less) is taken to asses model prediction accuracy. The Dev set informs the model, and the model parameters are tweaked until high accuracy on the Deve set is achieved.

(3) Test Set: Another small chunk of the original training set (20% or less) is taken to quantify the general accuracy of the model. This data set *does not* inform the model.

The motivation behind splitting up the training set comes from reasoning over how to quantify and increase the accuracy of machine learning algorithms. Further information on Train/Dev/Test sets is left to the reader to explore.

To arrive at the mathematical theory of neural networks, however, we must take regression into the realm of *classification*, which is the name given to logistic regression algorithms with an enforced decision boundary that forces the prediction into one or more discrete categories. The simplest classification problems are so-called binary classification problems, such as:

(1) Tumor Classification: is the tumor in a medical image malignant or benign?

(2) Atmospheric Feature Classification: is there a polar mesospheric cloud in this data set or not?

Somewhat unlike univariate regression problems, these simple binary classification problems have significant areas of application. Complex, or multi-class classification problems include:

(1) Number/Letter Classification: Is this hand written number 0, 1, 2, ..., 999, or 1000? Is this hand written letter A, a, B, b, C, c, ..., or z?

(2) Car Eyesight Classification: Is this a stop sign? Is this a pedestrian? Is this a dog? Is this a green light, yellow light, or red light?

As an interesting aside, multi-class classification is what our brains do incredibly well, even to the point where what we value in a particular instance determines how we classify an object in our environment. For example, if a person is walking through the forest and they see a cave, they may classify the cave as just a cave; however, if that same person is running from something in the forest that same cave may now become classified as a "bad/good place to hide." Although seemingly trivial, this value-based classification ability is a demonstration of profoundly complex intelligence.

The alternative name for classification, as mentioned earlier, is logistic regression. This name is informative as it alludes to the process by which regression algorithms become classification problems: classification is possible when the regression model is passed through the logistic equation. The logistic equation, or Sigmoid function, is a non-linear function which intakes the model parameters and inputs, and returns a probability. It is given by the following:

$$\sigma(x^{(i)}) = \frac{1}{1 + e^{-\theta^T x^{(i)}}} \tag{2.66}$$

A decision boundary is imposed on the output probability, which forces the output into a discrete category, represented by integer values. This output is the hypothesis of the classification algorithm.

$$\hat{y}(x^{(i)}) < 0.5, \quad y^{(i)} \to 0 \tag{2.67}$$

$$\hat{y}(x^{(i)}) \geq 0.5, \quad y^{(i)} \to 1 \tag{2.68}$$

Similar to the standard regression algorithm, classification problems are solved as optimization problems using gradient descent with the following loss function, the so-called **cross-entropy loss function**:

$$\mathcal{L}(\theta) = -\frac{1}{m}\sum_{i}^{m}\left[y^{(i)}log(\hat{y}(x^{(i)})) + (1-y^{(i)})log(1-\hat{y}(x^{(i)}))\right] \tag{2.69}$$

If one were to plot the two terms in the above loss function, it would be evident that this function is strictly convex, thus guaranteeing a unique solution to the optimization problem characterizing classification problems:

$$\min_{\theta} \; -\frac{1}{m}\sum_{i}^{m}\left[y^{(i)}log(\hat{y}(x^{(i)})) + (1-y^{(i)})log(1-\hat{y}(x^{(i)}))\right] \tag{2.70}$$

We now have arrived at the point of departure into the mathematical foundations of standard artificial neural networks, the understanding of which is necessary for subsequent material on more complex neural networks such as the encoder-decoder convolutional networks used in this thesis.

The quantitative desire for neural network algorithms is motivated by the following question: what if the hypothesis we are trying to learn is highly non-linear? For example, to make compute a regression model predicting what is being seen in the image captured by a 1000x1000 pixel color image would require a model parameter space of over 3 billion parameters. This represents a color image taken by a 1 megapixel camera, which is – by today's standard – pathetic resolution. How then, do we deal with the size of these problems? The answer: neural networks, which can solve these problems accurately in significantly smaller model parameter spaces.

On the highest level, what a neuron does is take in an input, perform some kind of processing, and produce a corresponding output, as seen in Figure 2.3.

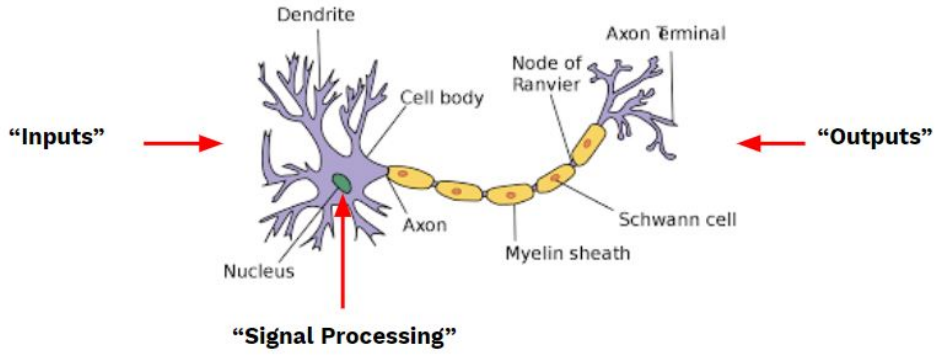

Figure 2.3: High level description of the functioning of a neuron.

The mathematical equivalent of this neuron is then given by the following:

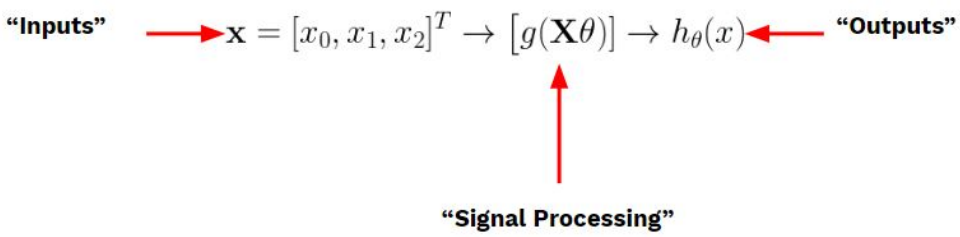

Figure 2.4: High level description of a mathematical neuron, the "perceptron," with h_θ representing the hypothesis, denoted $\hat{y}$ elsewhere in this thesis, and $g()$ representing the sigmoid or logistic function.

These neurons are then assembled into network architectures, as seen in Figure 2.4. At this point, conceptual understanding of what the network architecture is actually performing is crucial to understanding why the neural network architecture is so effective at learning complex hypotheses. Each neuron in the input layer, that is the first layer from left to right, is running a logistic regression optimization problem to create a hypothesis from the input data set. In order to accomplish this effectively, the network *needs* to be randomly initialized. In terms of multivariate calculus, the first layer of the network is receiving as its input a non-linear parameter space with as many dimensions as there are model parameters or weights for that layer, and each neuron is running gradient descent to find *a single* minima of the surface created by plotting the cost function against each model parameter. With random initialization, each neuron is hopefully

arriving at a different minima of this function, which is in essence what is giving these networks the ability, initially, to recognize many patterns within a data set. The subsequent layers are then establishing the relationship between the outputs of the neuron layer before it, and running the same optimization problem – finding the minima of their cost functions plotted against the *outputs* of the previous layer. In actuality, the network is only performing one cost function calculation, but the algorithms with which the network is trained, which will be discussed shortly, allow each layer of weights in the network to understand how much they are contributing to the overall cost of the network.

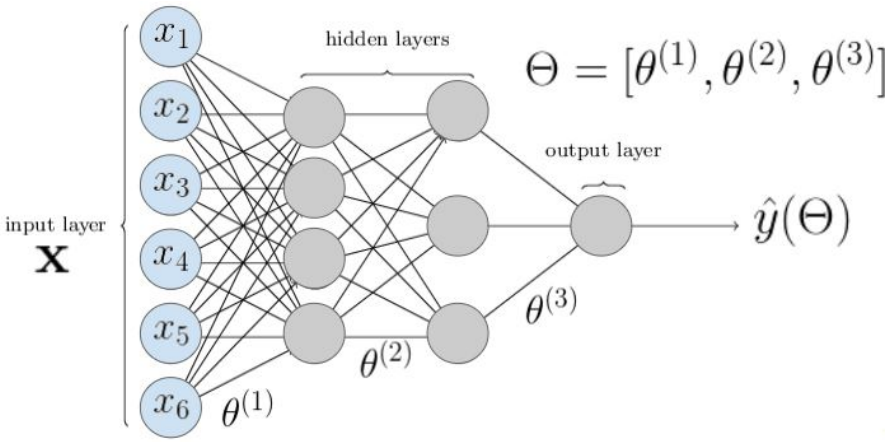

Figure 2.5: Standard Artificial Neural network architecture, a multi-layer "perceptron", with $\hat{y}(\Theta)$ representing the hypothesis of the network.

In the context of computer vision and convolutional networks, what these layers of the networks are learning is capable of being visualized. The first layers learn the simplest patterns recognizable: straight lines at different angles. The second layer combines the straight lines from the first layer to identify the next-most complex pattern in the image, such as curved lines. The following layer learns to identify more complex patterns in the image that can be built out of straight and curved lines. The following layer may then learn to recognize complex objects composed of the patterns learned by the previous layer. This example motivates the importance of the *depth* of neural networks and the subsequent buzzword "deep learning." On a computational level, depth is also advantageous: a deep network actually has a significantly smaller problem to solve than a

shallow network. This is a profoundly interesting phenomena, indicating that the geometry of a network is significantly more important than the shear number of neurons.

A high-level description of the training process is given as follows. A network is trained through a combination of two mathematical "passes" through the architecture. The so-called *Forward Propagation* pass is the first step in training, where an input training example is fed into the network, and the network produces an initial hypothesis. This hypothesis is compared with the known output, yielding the initial cost. This cost is then utilized in the second pass through the network, the so-called *Back Propagation* pass, which calculates the partial derivative of the loss function with respect to each weight of the network. The weights of each layer are then adjusted within the gradient descent algorithm using the partial derivative information from the back propagation pass. This process of forward propagation and back propagation is then repeated for each training example in the training set, resulting in a "trained network." For example, if a network is being trained to recognize handwritten digits 0-9, the training set would be composed of, say, 10,000 examples of handwritten digits for each number 0-9. The network would be running forward propagation and back propagation on each of the 10,000 training examples for each number 0-9 as it adjusts the weights to maximize its ability to recognize the digits (minimizing the number of times it predicts incorrectly).

Excellent in-depth, quantitative explanations of the training algorithms are widely available online, and the mathematical details of training are left to the reader to explore if they wish.

The final preliminary mathematical notions of importance are that of *activation functions* and *bias*. The sigmoid or logistic function is but one of a wide range of non-linear functions which have a two fold purpose: they give a value representing the so-called *activation* of a neuron, and they transform the neuron's linear model prediction onto a non-linear function. If neurons did not use a non-linear activation function, the neural network would simply be a computationally expensive linear regression algorithm. With respect to the first listed purpose of activation functions – on the highest level, a neuron's level of activation corresponds to the magnitude of the numerical value inside of it. If the activation function is the logistic function, the neuron activates to a number

between 0 and 1: the number 0 denotes no activation, and 1 denotes full activation. The *tanh* function has been explored as a scaled version of the sigmoid function that is centered around zero, but the most widely used activation functions in deep learning today are the variations of the so-called Rectified Linear Unit (ReLU) functions. The standard ReLU is defined by the following function:

$$ReLU(x) = max\{0, x\} \tag{2.71}$$

The ReLU function yields a few desirable behaviors. The function is non-linear, but it has a linear slope for positive activation values, unlike the sigmoid or *tanh* functions, whose gradient approaches zero at their extremes. This behavior of sigmoid or *tanh* activation functions lead to the "vanishing gradient" problem, in which the gradient calculations during training converge to zero, resulting in an overall convergence failure of the network. Due to its linearity for positive x-values, the ReLU function does not suffer from the problem of vanishing gradients, and provides substantially higher convergence rates than the sigmoid or *tanh* functions. The ReLU function also maps all negative activation values to zero, resulting in a network behavior called "sparse activation," in which only the positively activated neurons show non-zero activation for any given input. An important note is that the ReLU function does not provide an output which is useful in a probablistic sense. As such, ReLU activations are only used in the interior or "hidden" layers of a network, and the sigmoid or the softmax (sigmoid for multiple outputs) activations are conventionally used in the output layer of deep networks.

Recall that *bias* is the second and final important notion behind neural networks. The bias, denoted by a vector of scalar values b, regulate the magnitude of activation of a neuron to encourage confidence. The prediction of a network with the bias term added in looks as follows:

$$\hat{y} = \sigma(\theta^T X + b) \tag{2.72}$$

Let us explore a quick example to understand the importance of the bias term. Consider a computer vision problem in which a particular neuron has converged on small curved lines as its learned feature. If the first term $\theta^T X$ yields a value of 12, a bias term of say, -11, will "encourage"

the neuron to be more confident in its prediction by *biasing* its output to a lower value. Consider the following example which the activation of two neurons, one with an unbiased prediction of 12 and one with an unbiased prediction of 22, are compared with and without a bias of -11:

$$\hat{y} = \sigma(12) = 0.999 \tag{2.73}$$

$$\hat{y} = \sigma(12 - 11) = 0.73 \tag{2.74}$$

$$\hat{y} = \sigma(22) = 0.999 \tag{2.75}$$

$$\hat{y} = \sigma(22 - 11) = 0.999 \tag{2.76}$$

We see from the above example that the bias term leaves the confident neurons highly activated, while reducing the activation of what may be described as "intermediately" confident neurons.

2.2.1 Convolutional Neural Networks (ConvNets)

Convolutional neural networks, or ConvNets as they are called in short, are a modified version of the standard neural network discussed in the previous section, and they present a highly desirable potential for solving numerically massive problems. Let us explore the workings of ConvNets to illuminate why this is the case.

Consider the problem of computer vision in which we seek to identify features in a $9x9$ pixel grayscale image. The input to the network is a $9x9$ matrix of pixel brightness values, as seen in Figure 2.6:

	0.7	0.1	.01	0.9	0.47	0.56	0.2	0.44	0.2
	0.99	0.5	0.22	0.56	0.88	0.2	0.75	0.4	0.02
	0.08	0.65	0.54	0.24	0.34	0.66	0.44	0.11	0.64
	0.26	0.74	0.9	0.27	0.46	0.44	0.98	0.91	0.16
9	0.29	0.23	0.54	0.59	0.04	0.85	0.31	0.29	0.74
	0.45	0.65	0.14	0.18	0.31	0.78	0.08	0.92	0.43
	0.44	0.31	0.51	0.51	0.82	0.79	0.81	0.53	0.35
	0.94	0.88	0.55	0.63	0.3	0.47	0.43	0.92	0.18
	0.94	0.97	0.44	0.59	0.61	0.26	0.44	0.65	0.11

9

Figure 2.6: Matrix representation of pixel brightness values, which serve as the input to a convolution layer.

Now, we "convolve" (note that this is very close to, but not technically speaking the same operation as a standard mathematical convolution) this input with a "filter" of arbitrary size; let's choose $3x3$ for the sake of simplicity, seen in Figure 2.7:

	0.7	0.1	.01	0.9	0.47	0.56	0.2	0.44	0.2
	0.99	0.5	0.22	0.56	0.88	0.2	0.75	0.4	0.02
	0.08	0.65	0.54	0.24	0.34	0.66	0.44	0.11	0.64
	0.26	0.74	0.9	0.27	0.46	0.44	0.98	0.91	0.16
9	0.29	0.23	0.54	0.59	0.04	0.85	0.31	0.29	0.74
	0.45	0.65	0.14	0.18	0.31	0.78	0.08	0.92	0.43
	0.44	0.31	0.51	0.51	0.82	0.79	0.81	0.53	0.35
	0.94	0.88	0.55	0.63	0.3	0.47	0.43	0.92	0.18
	0.94	0.97	0.44	0.59	0.61	0.26	0.44	0.65	0.11

9

* 3

1	0	-1
1	0	-1
1	0	-1

3

Figure 2.7: Input matrix convolution with filter of size $3x3$.

The convolution step is an element-wise product-sum, yielding the following matrix for the first step:

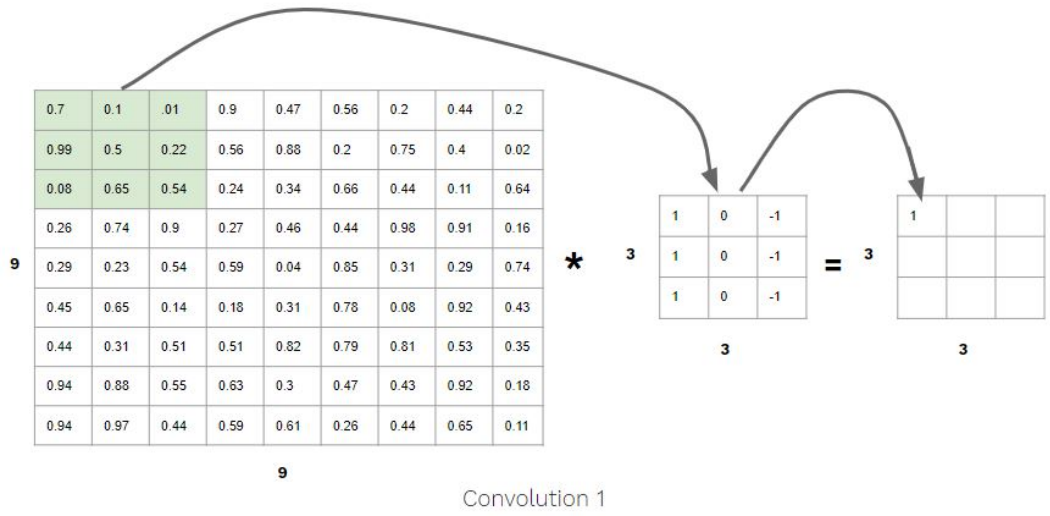

Figure 2.8: First convolution step.

The next step:

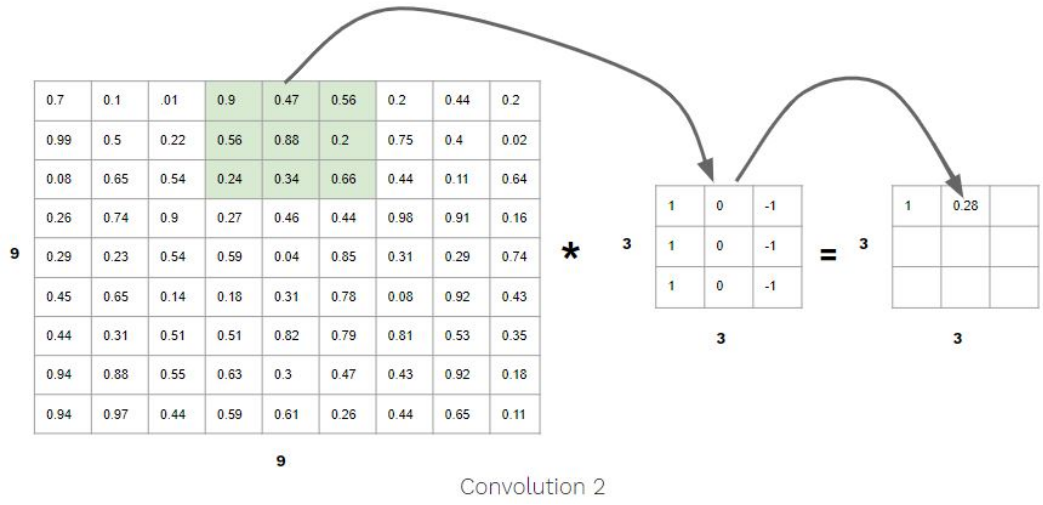

Figure 2.9: Second convolution step.

This process of stepping through the input matrix product-summing it's elements is continued, resulting in the following completed convolution:

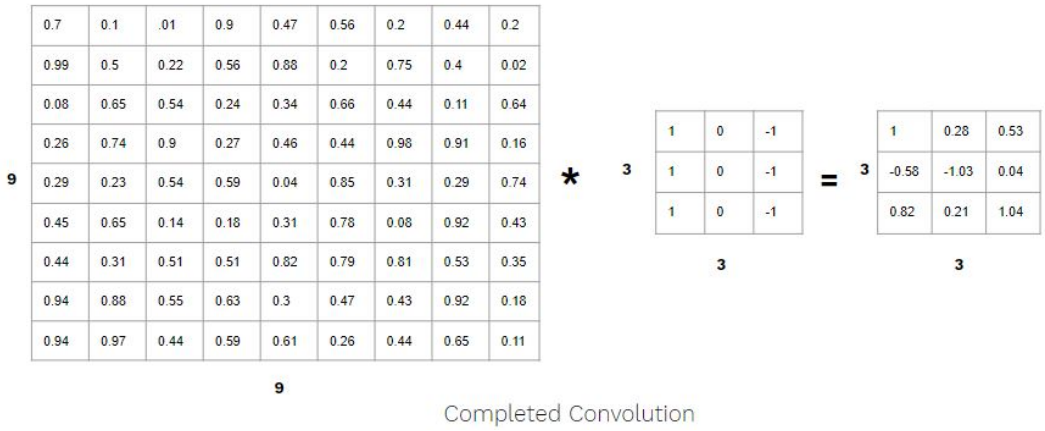

Figure 2.10: Completed Convolution.

What we have done here, in essence, is take the 81 element input matrix and represent it as an inverse convolution operation between the a filter matrix and the resulting output matrix. Here, we start to see the beginnings of convolution's potential for reducing data size by modifying the way in which it is represented.

Now, a convolutional neural network takes this convolution operation and tries to learn the *filter matrix element values*; in other words, the entries of the filter matrix are the model parameters. Consider now that the filter, or weight matrices, are represented by w, where w is a matrix of weight values w_{ij}. The problem the convolutional network is seeking to solve is to optimize the value of each w_{ij} such that the output cost of the whole network is minimized, as seen in Figure 2.11. An interesting side note is that the example filter given above is one of the possible filters for recognizing vertical lines in images.

Here is where we can see the advantage of a ConvNet for numerically massive problems, such as those seen in computer vision. Consider the problem where we seek to recognize features in a RBG color input image of size 1000x1000 pixels – a 1 Megapixel color image. This problem's input is now no longer a single matrix, but rather three matrices concatenated side by side into a 3-dimensional array, where each "slice" or "channel" – as they are called in ConvNet literature

– represents the pixel brightness for each of the 3 pixel types (R, B, G). This problem then has $1000 \times 1000 \times 3 = 3$ million input numbers. Let's say that we run this problem through a single layer ConvNet which has 10 filters, each of size $15x15$, and 10 subsequent biases. The ConvNet then has $15 \times 15 \times 10 + 10 = 2,260$ model parameters to learn. Notice that the number of model parameters in a ConvNet is actually *independent* of the input size. Now consider a standard neural network with one layer put to the same task. Let's assume this network has a single layer of 500 neurons and subsequently 1 bias term. This network then has $500 \times (1000 \times 1000 \times 3) + 1 = 1,500,000,001$ model parameters to learn. We see here that the convolutional network, due to its unique abilities to reduce data size, is uniquely posed to tackle numerically large problems.

As an interesting aside, and as a testament to the miraculous power of the human brain (and brains in general), note that the human eye sees in both color and grayscale (cone and rod cells) with a resolution of approximately 525 megapixels. That leads to an input "matrix" to the optical region of your brain of approximately $525000000 \times 3 \approx 1.75$ billion "numbers" or unique nerve signals. Your brain then processes all of this approximately 30 times per second, and is capable of cross-referencing real-time visual data with your optical memory to identify millions of unique features in your environment within milliseconds.

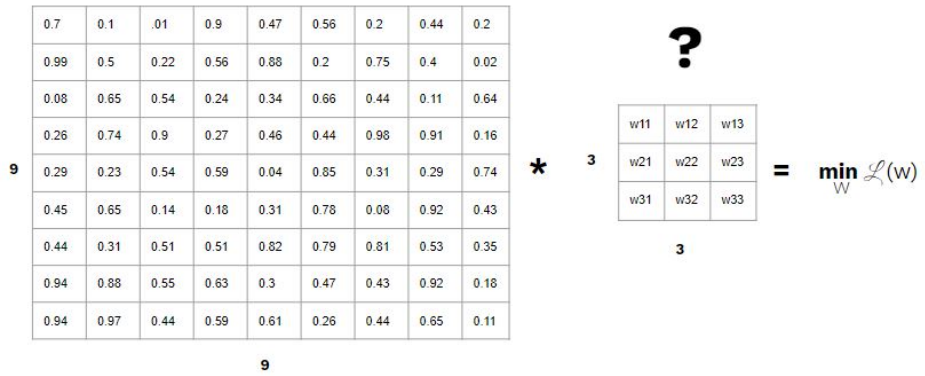

Figure 2.11: High-level description of ConvNet algorithm, which seeks to find optimal values of the filter matrix elements w_{ij} such that the ouput cost of the network is minimized.

To build a more complex ConvNet, this process of convolving a matrix with a filter is simply layered and then usually passed through one small standard neural network to classify the output. This final standard neural network is referred to as a "fully-connected" or FC layer in ConvNet literature.

Within a ConvNet, there are also multiple types of layers which serve to perform several useful functions, such as to further reduce the number of learned parameters. A "pooling layer" looks at a region of it's input and performs a simple operation to reduce that region to a single number. For example, a 3x3 "Maxpool" layer looks at a 3x3 region of its input matrix and outputs the largest value from that region.

As an example of a what a deep ConvNet looks like, consider the following image of the famous "LeNet-5" ConvNet [3], which is trained to recognize hand-written digits:

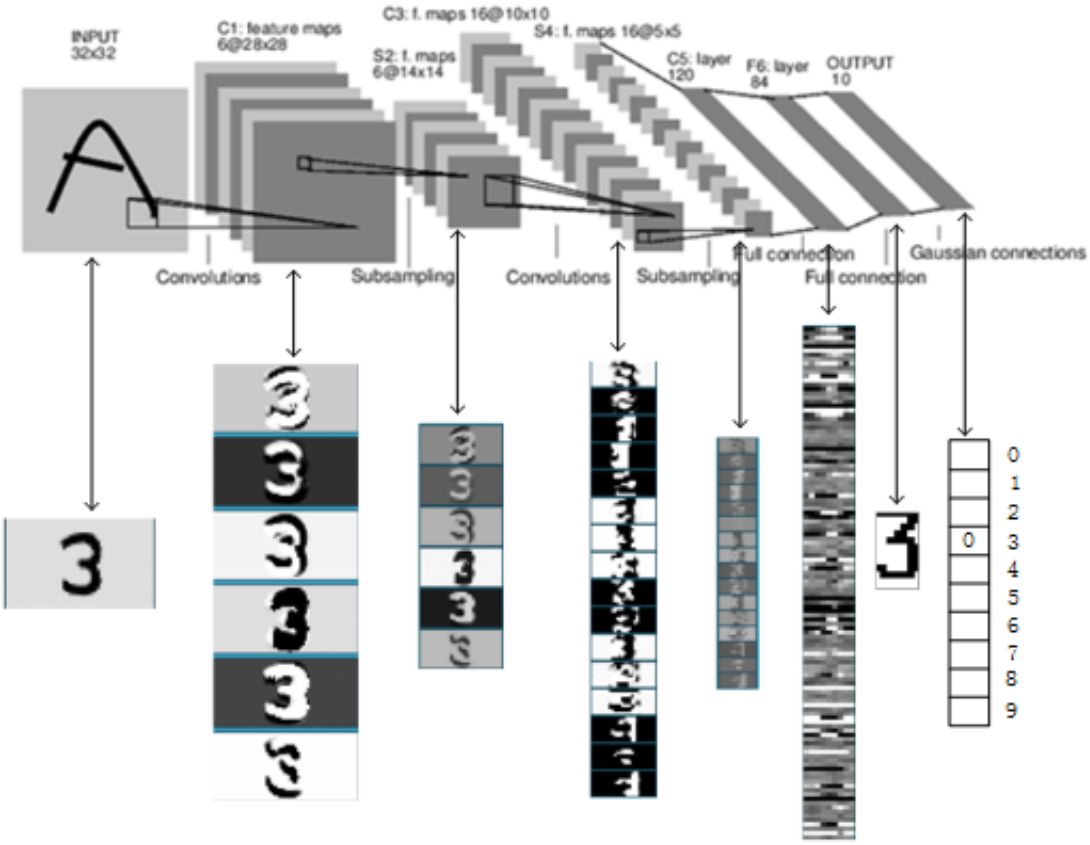

Figure 2.12: LeNet-5 [3], a deep convolutional network for recognizing handwritten digits.

2.2.1.1 Encoder-Decoder ConvNets

Encoder-Decoder Networks are a special ConvNet architecture which poses a very useful set of capabilites. To understand what an encoder-decoder network is and what kinds of capabilities they possess, let us go through an example of computer vision and discuss where each element – encoding and decoding – exercise their respective utility.

Consider the problem of training a convolutional network to learn how to recognize features in the image obtained from the cameras on a self-driving car. This network needs to learn how to take an input image and determine whether there are pedestrians, cars, buses, stop signs, lights, or other combinations several hundred other objects pertinent to driving located in the image. A convolutional architecture similar to that of LeNet-5 discussed in the previous section possesses the ability to recognize and classify objects in the image – it can say in a binary fashion, with a certain probability, that there either *is* or *is not* one or more of the objects it knows to recognize located in the image. This information, while necessary, is not sufficient to actually drive a car, however. Although the network has learned to recognize that there is indeed an object within the image, it provides absolutely no information as to *where* in the image that object is. In some sense, we can say that this ConvNet has *encoded* the feature information contained within the original image into a significantly size-reduced representation (i.e. its output prediction).

This begs the question: is it possible to take the encoded information about the objects found in the image and *decode* that information to find out where in the original image that object is located? In a more abstract sense, can we take the encoded information, combine it with data we have previously computed to generate an output containing useful information? The answer, of course, is yes. The "decoder" half of the encoder-decoder network acts like the original ConvNet in reverse: it takes the encoded data and, like pooling in reverse, "upsamples" the encoded data by combining it with data from the corresponding layer of the encoder half of the network. The decoder repeats this process, resulting in a reconstruction of the original image that contains the information about the objects contained within it. Figure 2.13 provides an example of this architecture:

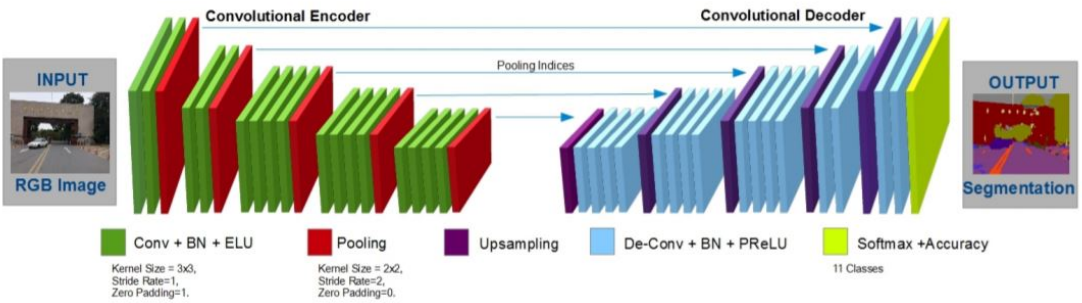

Figure 2.13: Encoder-Decoder Network Architecture for Road-Scene Feature Recognition [29]

In Figure 2.13, we see that the information from the pooling layers is carried across into the upsampling layer of the decoder, resulting in a output image from the network that contains the features recognized by the network, denoted by the different color regions. The problem type that this network is formulated to solve in this example is a so-called "Image Segmentation" problem.

As a side note, Encoder-Decoder networks are, in some sense, Generative Adversarial Networks (GANs) with the generator network and the discriminator network taking each other's usual location, respectively. The specific machinery of GANs will not be discussed in this thesis.

It should be noted that Encoder-Decoder networks have many powerful applications beyond the given computer vision example, such as those in the field of Natural Language Processing (NLP) with Recurrent Neural Networks (RNN's), which will not be discussed in this thesis. In this thesis, an encoder-decoder network is used to generate convergence predictions of topology optimization problems by taking in image-based representations of the solution gradient and treating the problem as an image segmentation task.

2.2.1.2 Inception Networks

Recall from the discussion of ConvNets that their filter size is independent of the input size. There is a certain amount of "hand-waving" that goes into the construction of a network architecture – where we have selected to have a filter of size 5x5, we may also have chosen to select a filter of size 3x3, or 9x9, or a pooling layer. The reasoning behind having a certain filter size is

motivated from a defining characteristic of most feature segmentation problems that convolutional networks are applied to: the salient features in a training set vary in size, location, and orientation, and some filter sizes are better for recognizing features of a certain size.

Given that each filters/layers provide their own distinct benefit when learning features in the input data, the following question becomes particularly interesting: why can't the architecture do them all? This is the motivating question behind the so-called "Inception Network." Inception networks perform several different operations on a *single* output from a given layer of the network and then concatenates the result from each operation into a new output. This method was pioneered by researchers from Google's DeepMind in [28]. Repeating these inception layers has been shown to have a powerful regularization effect on the network, leading to a network architecture which generally outperforms standard ConvNet architectures. An inception layer from Google's first inception network [28] is seen in the Figure 2.14.

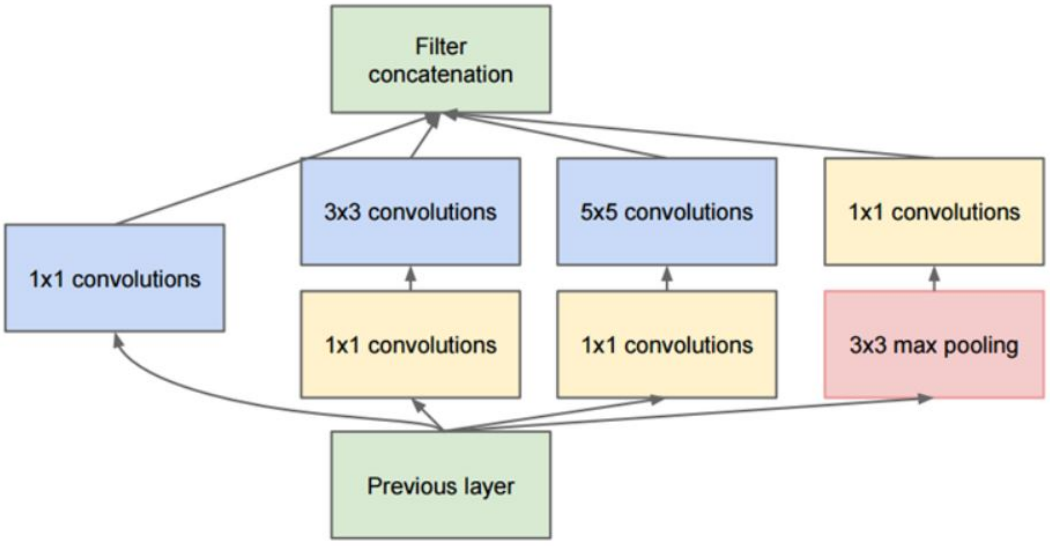

Figure 2.14: Inception Module from the GoogleNet inception network, adapted from [28].

Chapter 3

Methodology

With the prerequisite mathematical machinery sufficiently explained, we are now equipped to dive into the core research of this thesis, which is motivated by the following question: is it possible to use the "universal function approximation" capabilities of neural networks to predict the convergence of topology optimization solutions? In other words, is it possible to input early iterations of a topology optimization solution into a neural network and have it make accurate predictions as to what the solution space will look like at a further point in the convergence process?

The first experiment is to verify the results of Sosnovik and Oseledets (2017) [25] with a slight modification of the network presented in their work. The second experiment is to test a modified version of the network of experiment 1. Finally, the third experiment is to run the same problem through a novel network architecture: an inception encoder-decoder network. The results of the networks are then compared to determine which method provides a higher binary accuracy, and also to investigate the effects of the inception architecture on encoder-decoder-based segmentation problems.

3.1 Experiment 1: Sosnovik-Oseledets Network Reproduction

3.1.1 Overview

As the subsection title would suggest, this was a rather straightforward attempt to construct the same network as Sosnovik and Oseledets and achieve similar levels of accuracy, but using a

Keras-based implementation without resorting explicitly to TensorFlow.

3.1.2 Architecture

The convolutional network implemented by Sosnovik and Oseledets is an encoder-decoder convolutional network posed to solve the SIMP-based topology optimization convergence prediction problem by posing it as an image segmentation task. The input of this network is a two channel image, both grayscale, with the first being the explicit result of the topology optimization algorithm and the second being a gradient field computed by subtracting the densities of the aforementioned iteration with the densities of the previous iteration. Letting $X_{in,i}$ denote the i^{th} channel of the input for a given training example, we have:

$$X_{in,1} = X_n \tag{3.1}$$

$$X_{in,2} = X_n - X_{n-1} \tag{3.2}$$

The encoder-decoder network is constructed such that the output image from the network, after passing through an activation layer, is a grayscale density image with the same resolution as the input image that serves as the networks convergence prediction. Figure 3.1 shows the structure of this network. The network is composed of 6 convolutional layers, each with a filter size of 3x3 and fed into a ReLU activation function. The first two layers have 16 filters, the second two layers have 32 filters, and the final two layers have 64 filters. Two dropout layers (left to the reader to research) are included to regularize the network.

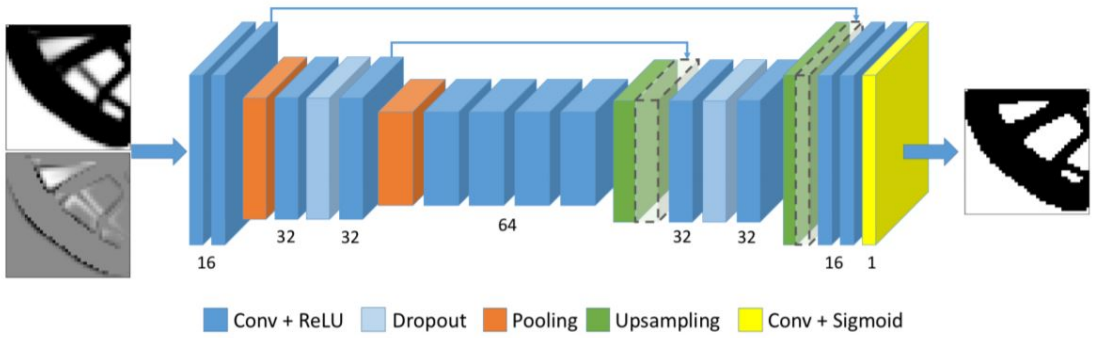

Figure 3.1: Encoder-Decoder network from Sosnovik and Oseledets (2017) [25].

The decoder section of the network reverses the encoder section, Upsampling where the encoder performed Maxpooling, and ending with a single filter that passes through a sigmoid activation function to yield the final prediction. The network architecture requires the input image be constructed such that the height and width are divisible by 4, and poses a parameter space of 192,113 parameters to learn.

3.1.3 Dataset

Sosnovik and Oseledets used the open source SIMP-based Topology Optimization software *Topy* (developed by [14]) to generate an initial set of 10,000 training examples for the network. Each training example in the training set contains 100 iterations from the *Topy* solver, yielding a total dataset with approximately 40,000 100-iteration training examples after rotating and reflecting each image.

3.1.4 Training Parameters

Optimizer	Adam
Loss Function	Binary Cross Entropy
Accuracy Metric	Binary Accuracy
Number of Training Examples, m	40,000
Training Epochs	15
Validation Split	0.05
Batch Size	10
Input Size	(40, 40, 2)

Table 3.1: Experiment 1 Network Training Parameters

3.2 Experiment 2: Sosnovik *et al* Network with Modified Input

3.2.1 Overview

This experiment implements the same network as Sosnovik and Oseledets shown in Figure 3.1, but trained with a 3-channel input with the hypothesis that it will yield a higher prediction accuracy from the network.

The reasoning behind inputting three iterations as opposed to one iteration and its gradient is that the additional input information may allow the network to make a more informed prediction. Additionally, the network now has the potential to *learn* the gradient from the density differences between the three inputs. Variations in the input, such as two or three iterations *and* their respective gradients is certainly a topic for further research.

3.2.2 Architecture

The input to the network has been slightly modified from that which is presented by Sosnovik and Oseledets, who input the two channel grayscale image described above. In this experiment, the input is modified to be a 3 channel grayscale image with each channel being an early iteration in the optimization process, specifically iterations 2, 4, and 6. A nuance of the network is that the inputs must also be organized in ascending order as listed below:

$$Xin, 1 = X_2 \tag{3.3}$$

$$Xin, 2 = X_4 \tag{3.4}$$

$$Xin, 3 = X_6 \tag{3.5}$$

The reasoning behind selecting these particular iterations is somewhat arbitrary. Some uniform criteria is needed for selecting the input to the network, and the question motivating the research is focused around whether the network can "look" at the early results from the topology optimization algorithm and predict where the solution space is headed. As such, these specific iterations were selected as the criteria since that results in the data set converged.

The network architecture is seen in Figure 3.2

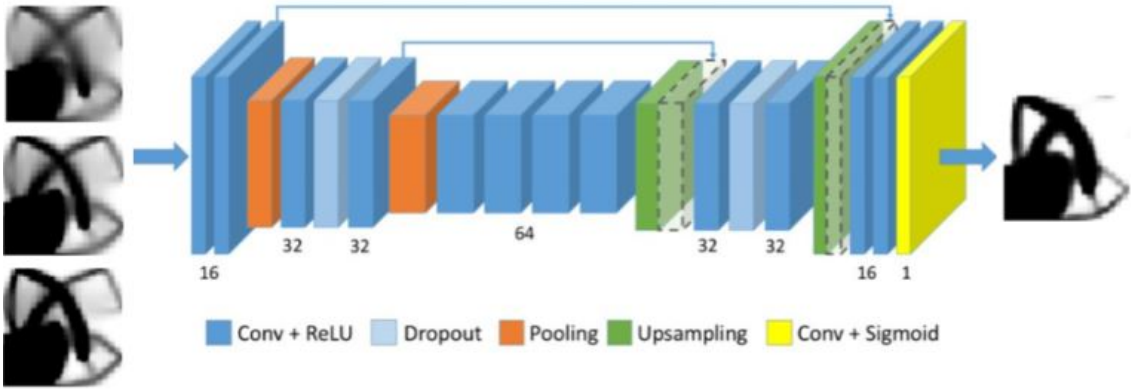

Figure 3.2: Modified Encoder-Decoder network from Sosnovik and Oseledets (2017) [25].

3.2.3 Dataset

The dataset used to train this network is identical to that of Experiment 1.

Optimizer	Adam
Loss Function	Binary Cross Entropy
Accuracy Metric	Binary Accuracy
Number of Training Examples, m	40,000
Training Epochs	15
Validation Split	0.05
Batch Size	10
Input Size	(40, 40, 3)

Table 3.2: Experiment 2 Network Training Parameters

3.2.4 Training Parameters

3.3 Experiment 3: Inception-Based Encoder-Decoder Network

3.3.1 Overview

The Inception-based encoder-decoder network is a novel network architecture implemented in this thesis to explore whether the benefits of inception networks transfer when inserted as elements of encoder-decoder networks. The reasoning behind formulating this architecture was two-fold:

(1) Well-made inception networks generally outperform standard convolutional networks when applied to the same problem. This motivates the following question: can inception encoder-decoder networks outperform standard encoder-decoder networks when applied to the same problem?

(2) Curiosity: Is it even possible to hybridize these networks? What unique challenges, if any, does this novel network architecture pose? If this network architecture appears to be more successful at predicting solution convergence, what tweaks can be made to the architecture to further increase the accuracy?

3.3.2 Architecture

The inception-based encoder-decoder network, as the name would imply, is a hybrid cross-over between a standard encoder-decoder network – as presented in the previous section – and an

inception network. A graphical representation of the network is seen in Figure 3.3.

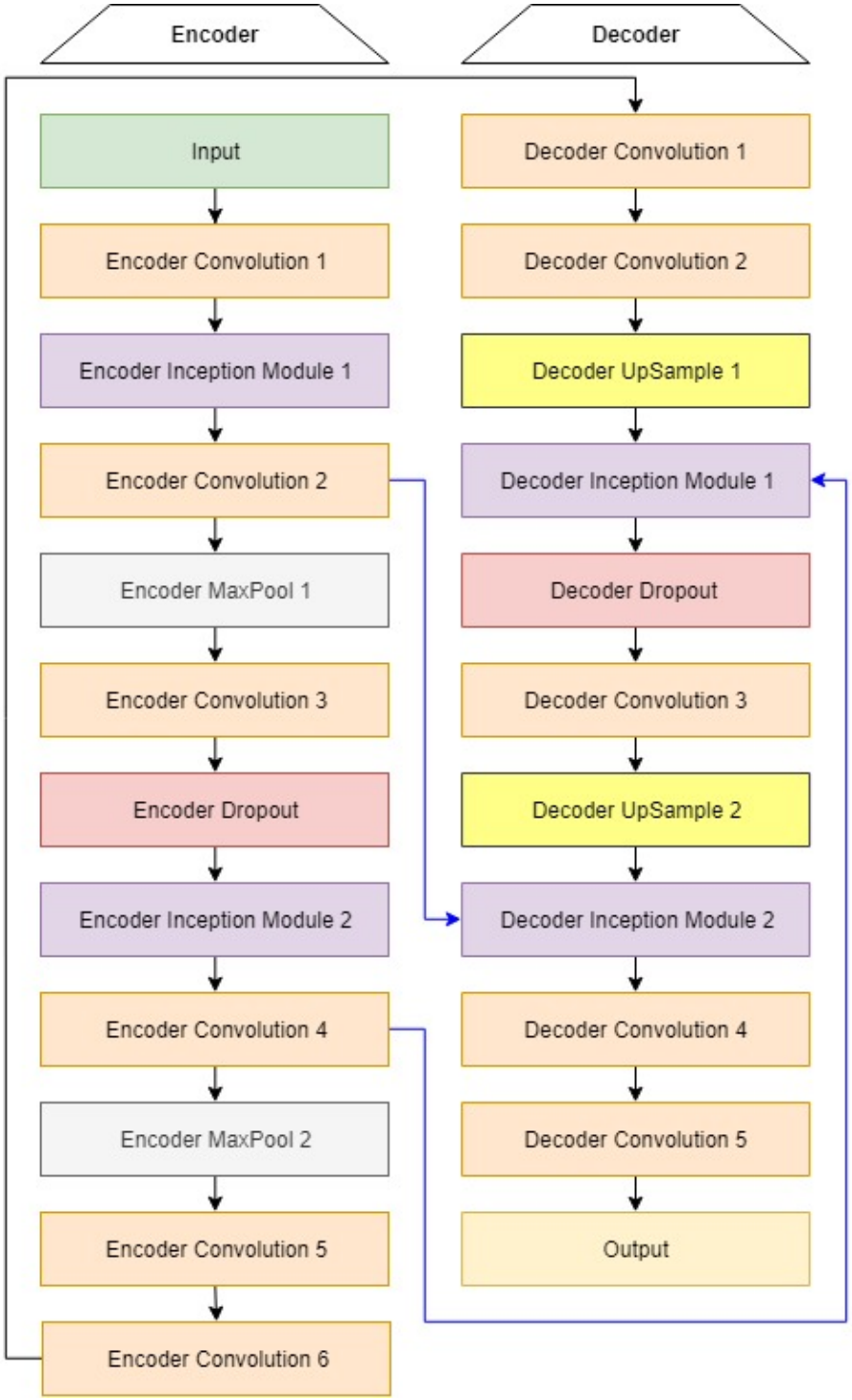

Figure 3.3: Hybrid Inception Encoder-Decoder network architecture.

The architecture of the inception module is identical to the inception module with expanded filter banks presented by Christian Szegedy *et al* in [27], seen in Figure 3.4.

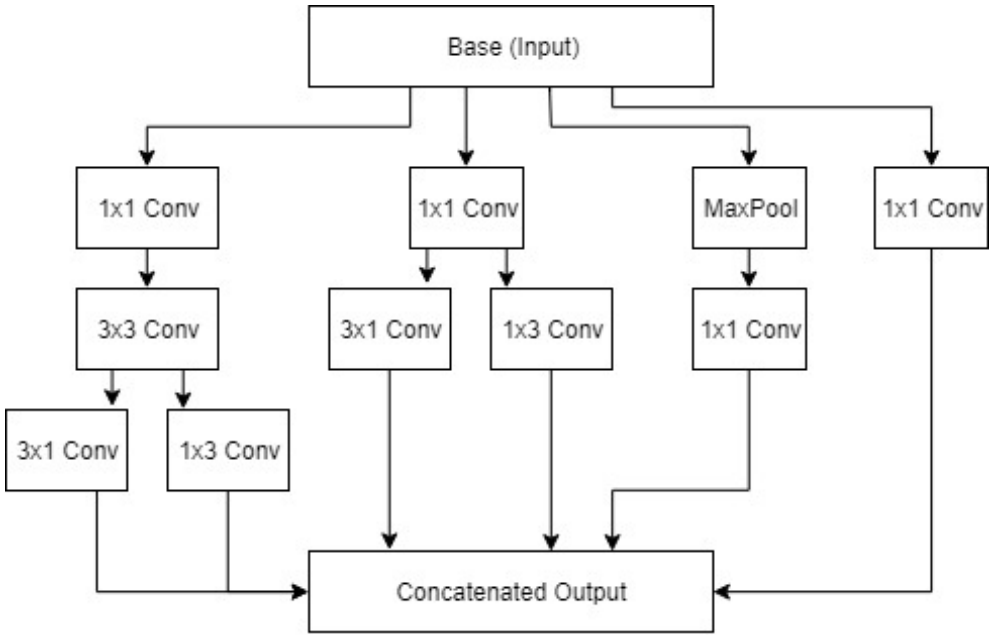

Figure 3.4: Inception Module adopted from [27].

Notice that the left-most tower and the tower directly next to it contain a rather strange convolution, that being the parallel 3x1 and 1x3 convolutions. The research done by Szegedy *et al* determine that two parallel 3x1 and 1x3 convolutions were actually more computationally efficient than a single 3x3 convolution. An additional nuance of this inception module is the stacked 3x3 to 3x1 and 1x3 convolutions of the left-most tower. This structure has replaced a single 5x5 convolution for the same reason as was mentioned above – Szegedy *et al* determined it was more computationally efficient than a single 5x5 convolution.

3.3.3 Dataset

The dataset used in this experiment is identical to that of Experiment 1.

3.3.4 Training Parameters

Optimizer	Adam
Loss Function	Binary Cross Entropy
Accuracy Metric	Binary Accuracy
Number of Training Examples, m	40,000
Training Epochs	15
Validation Split	0.05
Batch Size	10
Input Size	(40, 40, 3)

Table 3.3: Experiment 3 Network Training Parameters

Chapter 4

Results and Discussion

4.1 Experiment 1: Sosnovik *et al* Network Reproduction

4.1.1 Training Results

Consider the following two figures:

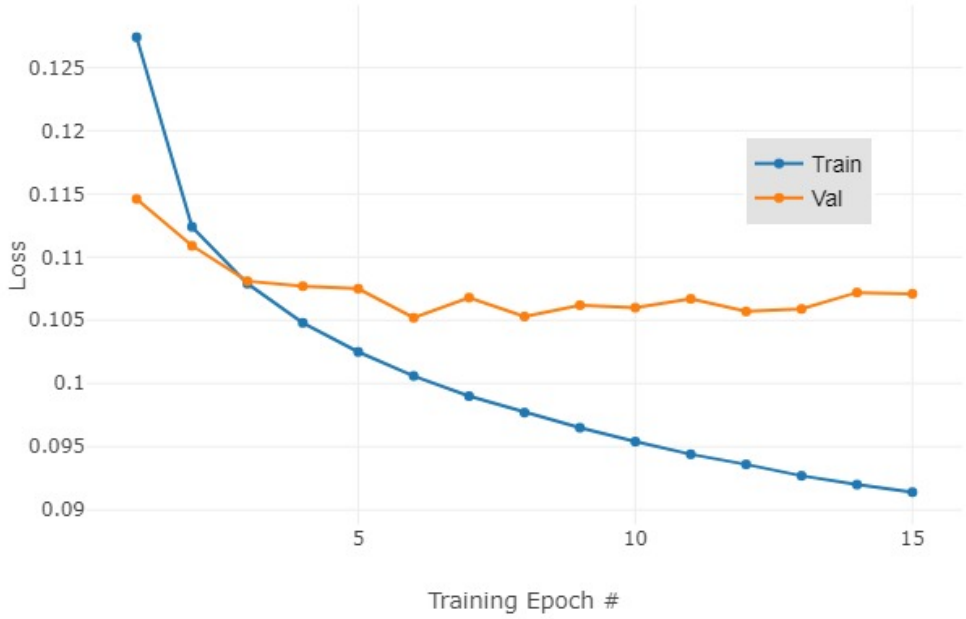

Figure 4.1: Training and Validation Loss of the Encoder-Decoder Network of Sosnovik *et al*.

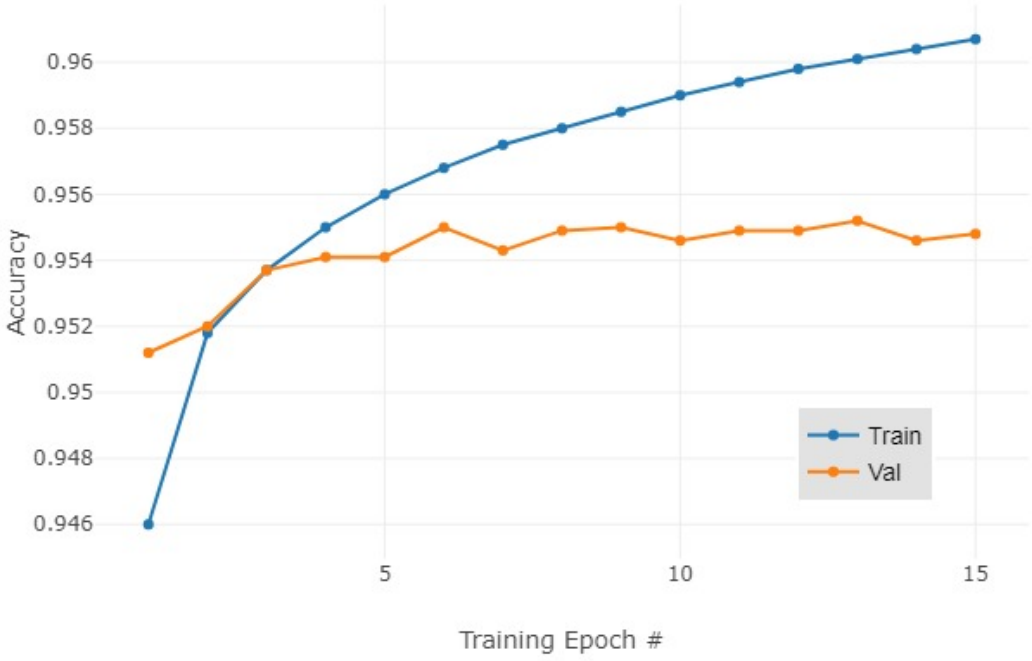

Figure 4.2: Training and Validation Binary Accuracy of the Encoder-Decoder Network of Sosnovik *et al.*

Figure 4.1 shows the cross-entropy loss and Figure 4.2 shows the binary accuracy of the network detailed in Experiment 1. Notice that the validation curves are beginning to plateau while the training set data continues to decrease. This behavior is indicative of one of two things: (1) the network is over-fitting the training set and consequently has a lower generalization accuracy, or (2) the network architecture is not optimally suited to solving this problem. This aspect of the networks performance was neglected in the results published by Sosnovik *et al*, and suggests this network is only capable of achieving a binary accuracy of greater than 99% – as published in [25] – on the training set.

This makes a great example as to the necessity of validation and test sets when analyzing a network's performance – the binary accuracy of the network on the training set is not the metric

with which to analyze its capabilities. Based on the figures above, the test accuracy can be predicted to stagnate below 96%, which is still indicative of a high-performing network, but not as high as reported by Sosnovik *et al* in [25].

4.1.2 Predictions

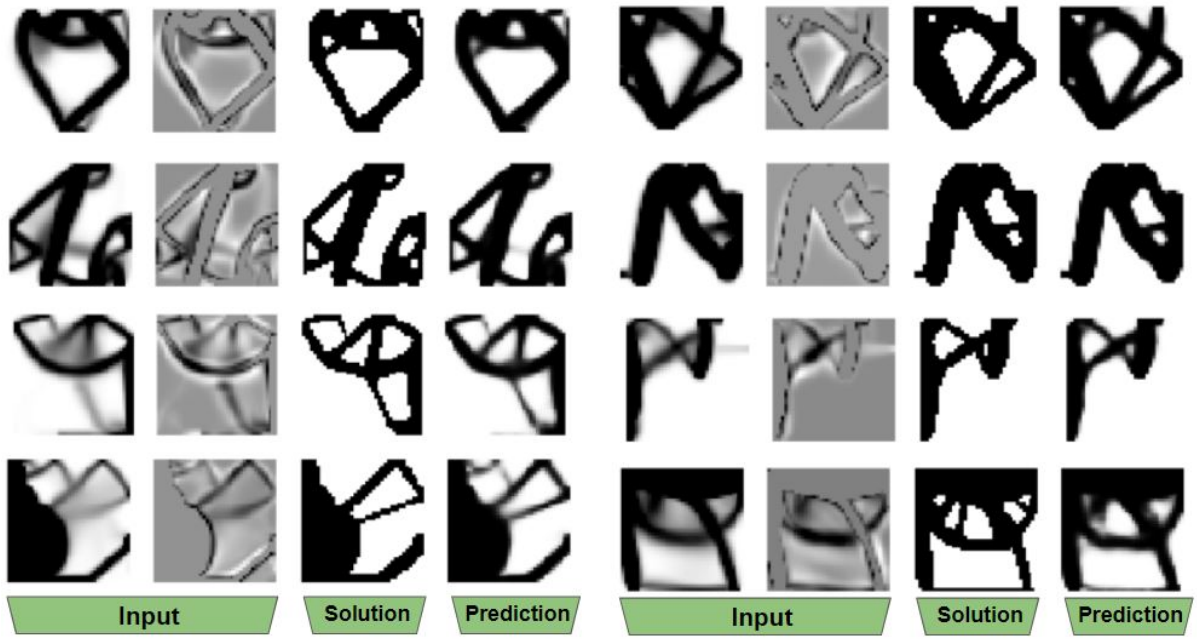

Figure 4.3: Network Prediction vs. Optimization Solver Solution for given input.

Figure 4.3 shows the prediction of the network alongside the solution from the topology optimization solver for a given set of inputs. The figure shows that, at least to the eye, the network does a fairly good job of making predictions, but seems to exhibit one particularly undesirable behavior: when the input to the network still contains intermittent densities, the network fails to resolve these regions in a way that eliminates certain members. Consider the example in the lower-left corner of the figure: this figure has a small member in the top of the image which is subsequently removed by the optimization solver, but which remains present in the prediction made by the network. The network also seems to struggle to resolve the image for when the inputs are higher in intermediate densities.

4.2 Experiment 2: Sosnovik *et al* Network with Modified Input

4.2.1 Training Results

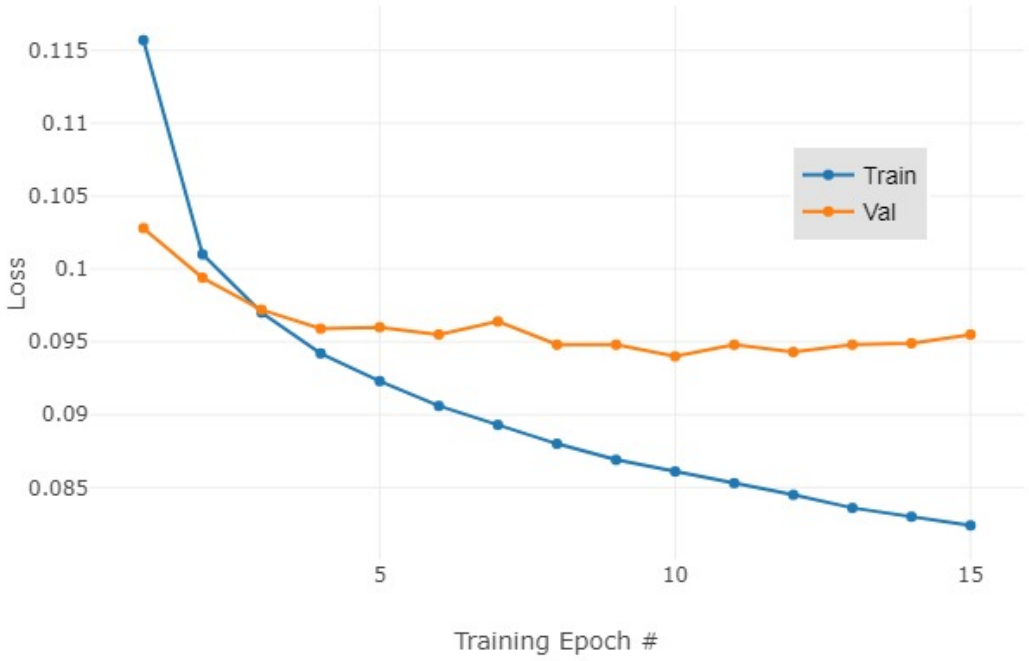

Figure 4.4: Training and Validation Cross-Entropy Loss of the Encoder-Decoder Network of Sosnovik *et al*.

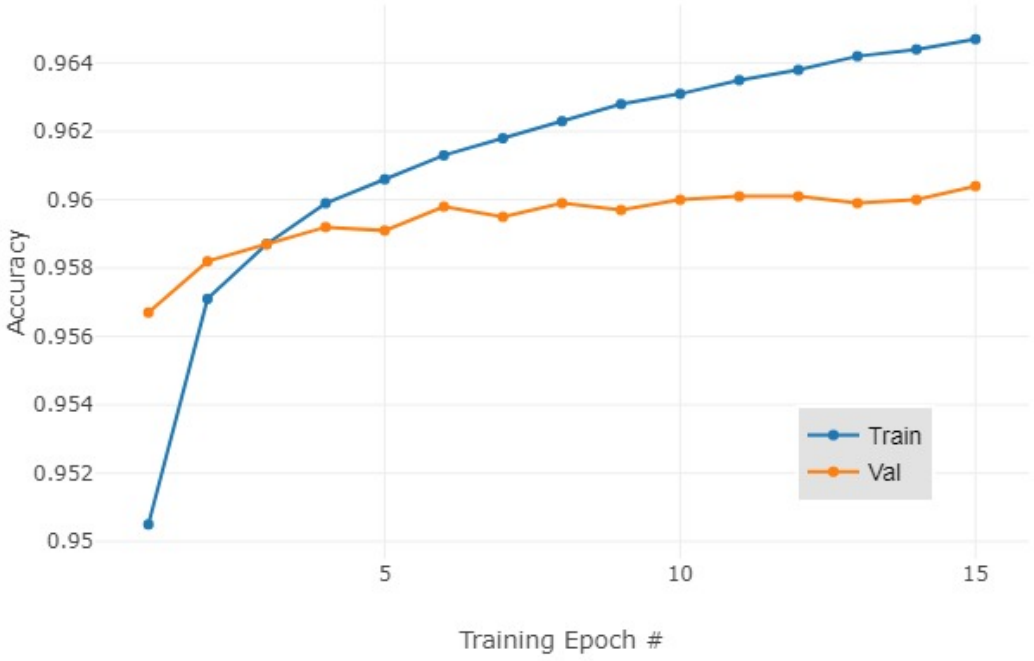

Figure 4.5: Training and Validation Binary Accuracy of the Encoder-Decoder Network of Sosnovik *et al.*

Figure 4.7 shows the cross-entropy loss and Figure 4.8 shows the binary accuracy of the network detailed in Experiment 2. The validation curve in both figures indicate that the modified input does not prevent the network from over-fitting the training set, and the behavior is somewhat consistent with that of the network in Experiment 1. It should be noted, however, that the validation accuracy of this network actually seems to be increasing slightly – at least over the given training epochs – indicating that it is over-fitting the data less than the network of experiment 1. This information permits a preliminary prediction that the network performs better with a 3-Channel input than with a 2-Channel input.

4.2.2 Predictions

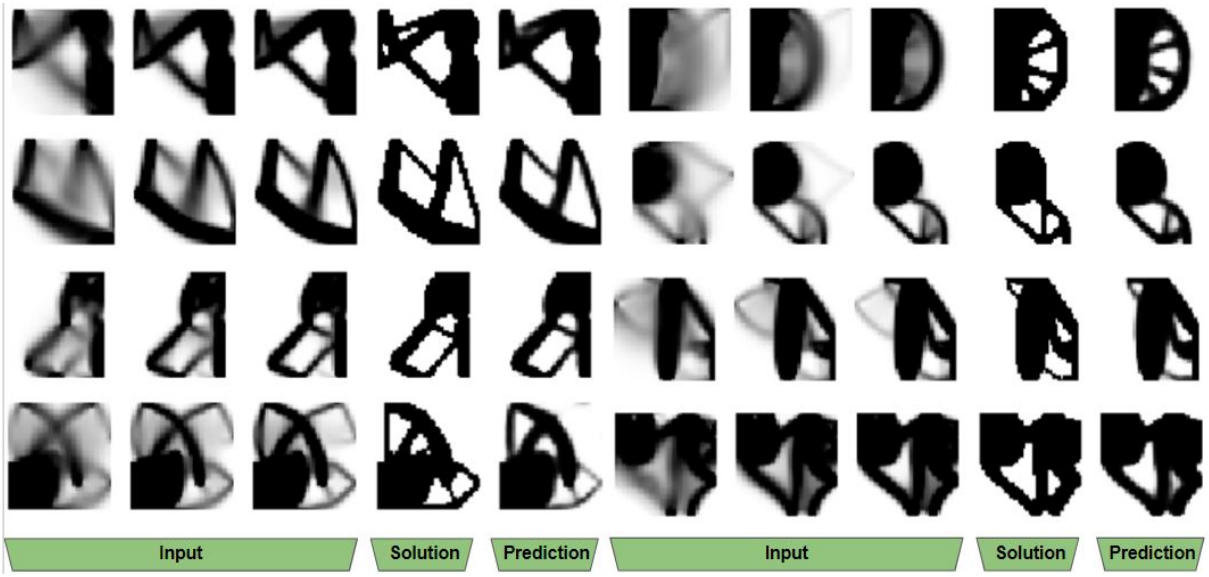

Figure 4.6: Network Prediction vs. Optimization Solver Solution for given input.

Figure 4.6 shows a collection of predictions from the network alongside the solution from the topology optimization solver for a given set of inputs. A very interesting difference between the results shown here and the results shown in Figure 4.3 is that this network *does* seem to resolve areas of intermediate density in a way that is consistent with the optimization solver. To see this, consider the example shown in the bottom-left of the figure.

4.3 Experiment 3: Inception Encoder-Decoder Network

4.3.1 Training Results

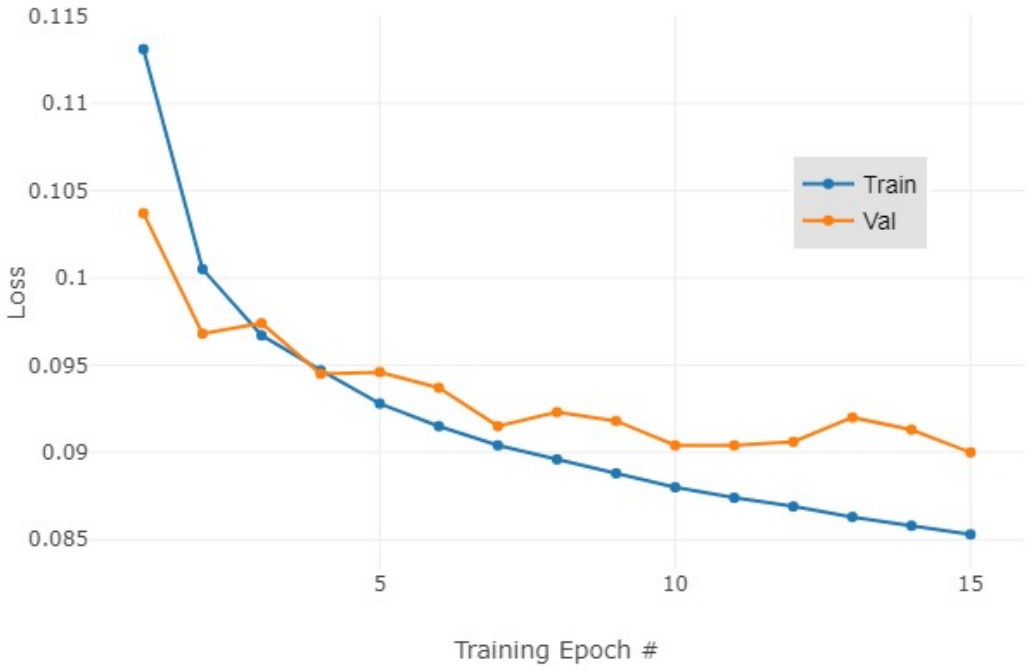

Figure 4.7: Training and Validation Cross-Entropy Loss of the Inception Encoder-Decoder Network.

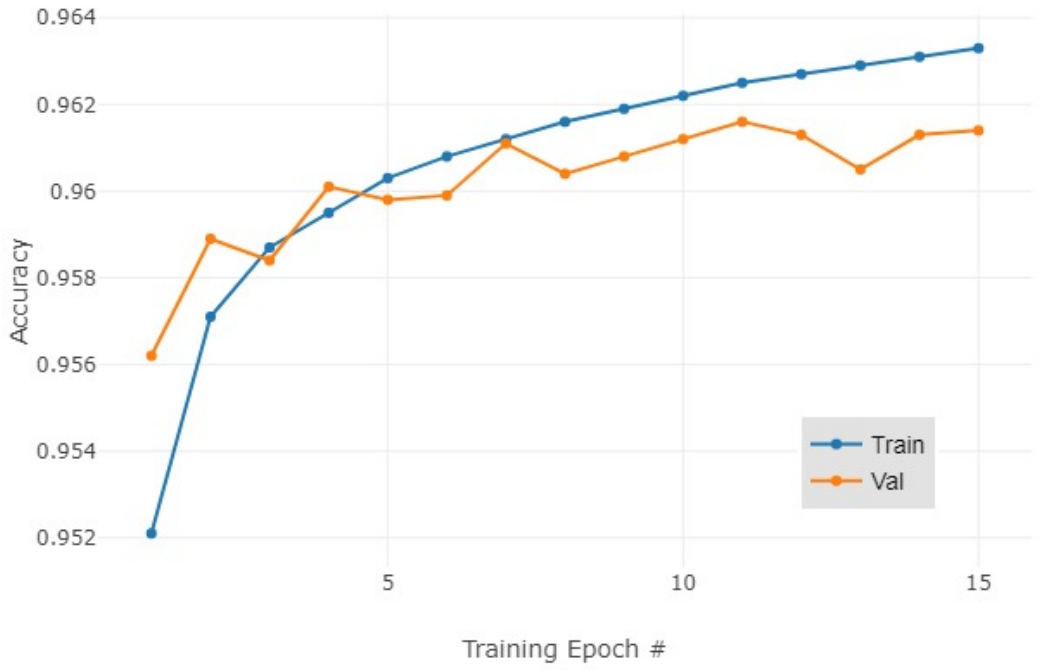

Figure 4.8: Training and Validation Binary Accuracy of the Inception Encoder-Decoder Network.

Figure 4.7 shows the cross-entropy loss and Figure 4.8 shows the binary accuracy of the Inception network detailed in Experiment 3. These figures present some very interesting behavior, namely that the validation curves seem to be steadily moving in the direction of the training curves. This is indicating that the inception network is fitting the data better (low over-fitting) than the networks of experiments 1 or 2. Figure 4.8 also indicates that the network achieves the highest accuracy of all 3 networks. The combination of high accuracy and low over-fitting permit the prediction that the inception network performs better than the other two networks.

4.3.2 Predictions

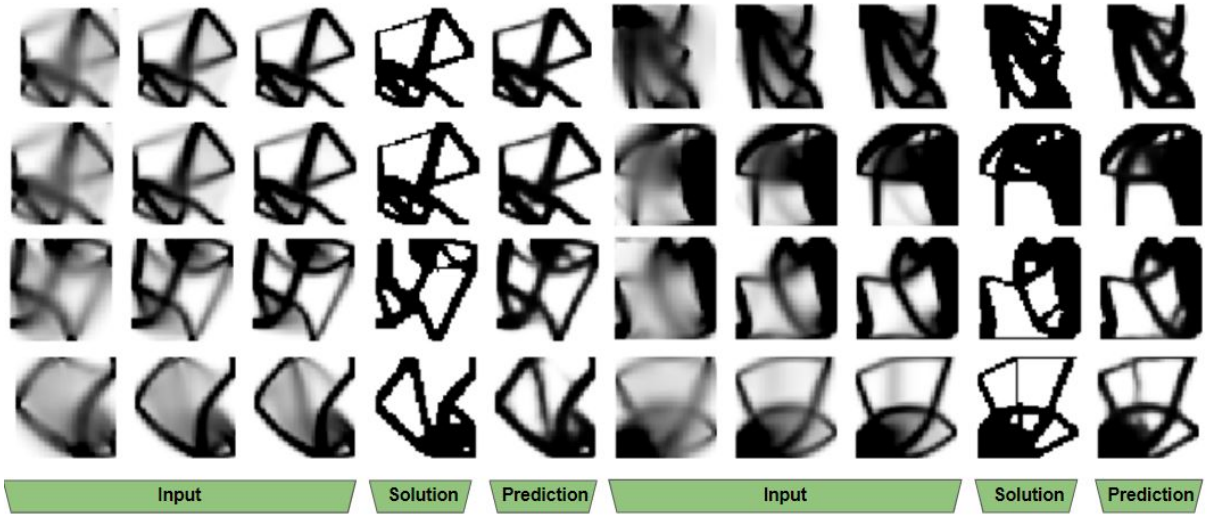

Figure 4.9: Network Prediction vs. Optimization Solver Solution for given input.

Figure 4.9 shows a collection of predictions from the inception network alongside the solution from the topology optimization solver for a given set of inputs. Similar to the network of experiment 2, it is apparent that the inception network seems to resolves intermediate density features, albeit slightly less well. An interesting aspect of the results to note is that this network structure seems to carry more of the intermediate densities into its final prediction – there's more "fuzziness" in the prediction. This is likely due to the fact that the encoding section of the network is retaining much more information about the inputs than the other two networks and is passing that information to the decoder for the prediction.

4.4 Experiment Comparison

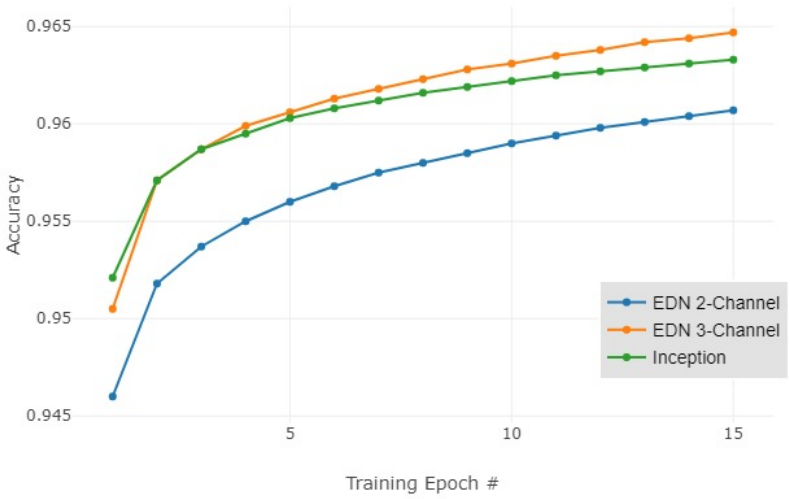

Figure 4.10: Training Accuracy Comparison.

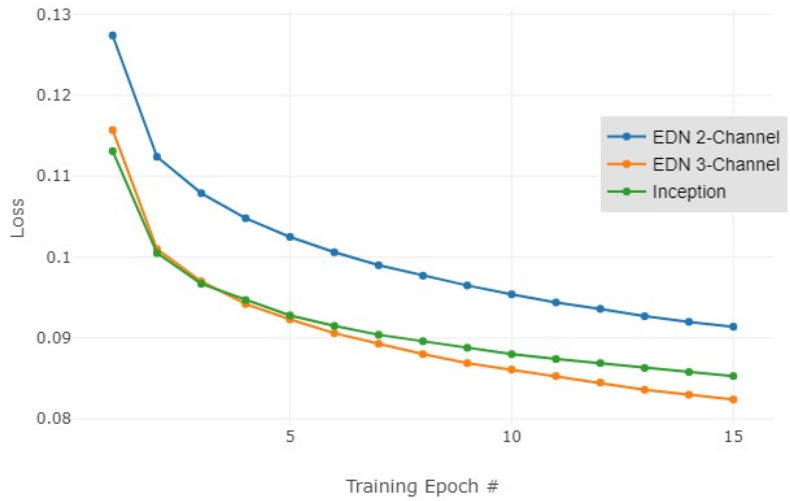

Figure 4.11: Training Loss Comparison.

Figures 4.10 and 4.11 show the training accuracy and loss of all 3 experiments, respectively. Notice that the network of experiment 2, denoted EDN 3-Channel in the legend, outperforms the other networks in terms of accuracy and loss. Recall, however, that performance on the training set is not a good indicator of which network generalizes the best to new data.

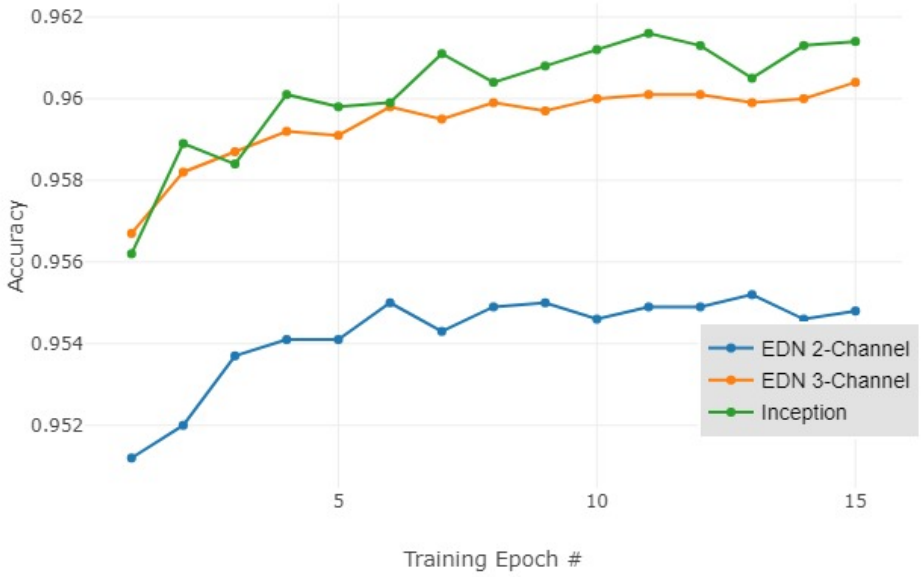

Figure 4.12: Validation Accuracy Comparison.

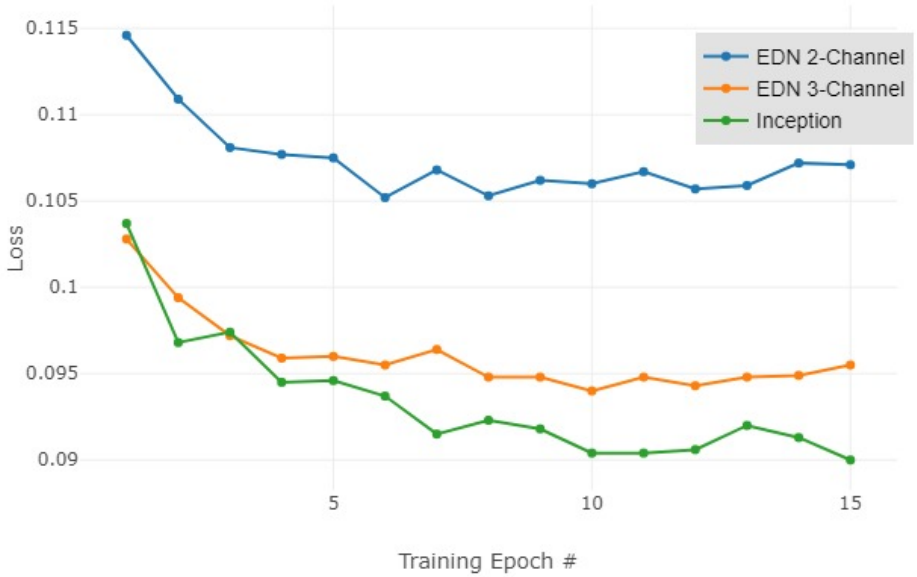

Figure 4.13: Validation Loss Comparison.

Figures 4.12 and 4.13 show the validation accuracy and loss of all 3 experiments, respectively. Note that both the inception network and the 3-Channel network show improved validation set performance, and both seem to continue to increase in accuracy throughout the given training epochs, whereas the validation accuracy of the 2-Channel network seems to stagnate.

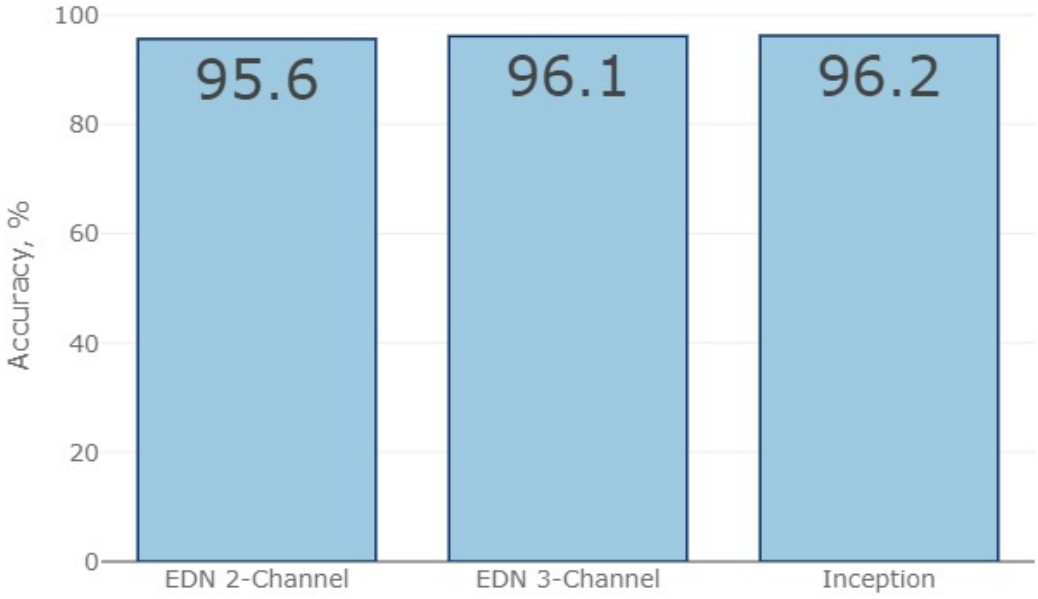

Figure 4.14: Test Accuracy Comparison.

Figure 4.14 shows the test set accuracy of the three networks, and serves to confirm the predictions made earlier about the overall performance of the networks. The 2-Channel network of Experiment 1 performs the worst out of all the networks, likely due to its over-fitting of the training data set. It is evident that simply by adding a third channel to the input, the network architecture of Sosnovik *et al* increases considerably in generalized accuracy.

Finally, the inception network outperformed all the networks on the test set, and based on its validation set metrics, can be predicted to continue to outperform the 3-channel network through further training epochs.

4.5 Discussion of Results

There are several important take-aways from the results of experiments detailed above:

(1) **Validation Sets**: It is of utmost importance to include information from the validation and test set when analyzing a network's performance. The failure to do so may deceive a researcher, scientist, or engineer using such a network into thinking it is performing at a much higher level than it actually is.

One should also be careful not to use information from the test set to analyze the generalization accuracy of the network when a validation set is not being used. Ideally, the validation set is used to inform the hyperparameters of the network, resulting in iterative changes to the network until optimal accuracy is achieved on the validation set. Only then does the test set provide insight into the generalized accuracy of the network. To put it simply, the test set should never inform the network, less it cease to perform the functions of a test set.

(2) **Network Comparison**: The 3-channel input provides notable performance improvements across all domains of interest when compared with the 2-channel input: it achieves a lower loss and higher accuracy on the training set, is less prone to overfitting the data set, and yields a higher accuracy on the test set. The inception network performs the highest of the network architectures presented, achieving the highest test set accuracy while demonstrating the best fit to the training data set. It should be noted that the inception network did have more learning parameters than the other two networks: the encoder-decoder architecture of experiments 1 and 2 contained 192,257 parameters and the inception network contained 223,985 – a 116.5% increase. That begs the question: could that be the reason why the inception network performed better than the 3-Channel encoder-decoder network? To some extent, yes; however, blindly adding more parameters to a network will in fact make the overfitting problem worse – the issue of overfitting is rather solved through regularization techniques and modifications to the network architecture.

With that said, it is expected that the inception network – because of its superior fitting behavior – would continue to outperform the 3-Channel encoder-decoder network even if they had the same number of parameters.

(3) **Applications to Topology Optimization**: The set of experiments confirms that encoder-decoder neural networks, posed to solve a segmentation problem, are capable of predicting topologically optimal solution convergence with a binary accuracy greater than 96%. The results clearly show, however, that the results from the networks contain regions of intermediate density and floating members that have rightfully been removed by the optimization solver. As such, these networks are not well-posed to "predict solutions," but rather to "predict solution *convergence*." These networks could greatly enhance the performance of topology optimization solvers if used to perform "leaps" through the solution space. Consider, for example, a solver algorithm which runs a few iterations of the topology optimization solver, feeds a selection of the early iterations into a convergence prediction network, passes the prediction from the network back into the optimization solver, lets the solver run several iterations, and then repeats the process. In this case, the "fuzzy" predictions of the network would be advantageous: they represent areas where the network has a lower prediction confidence and would hence allow these low-confidence regions to be solved by the mechanically-informed optimization solver.

Chapter 5

Conclusions and Future Work

5.1 Summary of Completed Work

In this thesis, the motivation for applying the universal function generation capabilities of neural networks to problems in topology optimization was laid out: neural networks contain the potential to learn the complex non-linear relationships between a design space and its topologically optimal solution, thus presenting a benefit of lowering the computational cost of solving such problems. A background of the mathematical theory behind the topology optimization solver was laid out, followed by a surface-level crash-course in the theory of neural networks.

The motivation was combined with the theory to pose a series of experiments: one which sought to investigate and validate the results of a previous research experiment; one which sought to investigate a hypothesis that modifying the input structure of the same network could improve its performance; and another which posed a novel hybrid Inception Encoder-Decoder network architecture to test and compare against the previous networks on the same segmentation problem.

The experiments were successfully carried out, demonstrating that the 3-Channel modified input successfully increased the performance of then network, both in terms of test set accuracy and the quality of the fit to the training data, and that the novel inception encoder-decoder network performed superior to the standard encoder-decoder network architectures.

5.2 Unanswered Questions and Future Research

The research in this thesis posed several interesting questions for further research:

(1) What modifications (regularization, architecture modifications, additional layers, etc.) to the standard and inception-based encoder-decoder networks increase their performance?

(2) What other problems is the inception encoder-decoder network posed to solve well?

(3) What are differences between what the inception encoder-decoder network is learning versus what the standard encoder-decoder network is learning?

(4) What are the limitations that the inception network runs into, given its current architecture?

(5) How do these networks perform when the solution they are attempting to predict is intermediate? To explain by example, how do they perform if applied to a problem which takes a GCMMA density-based solver 2500 iterations to solve, tasked with taking iterations 20, 22, and 24 as inputs to predict what the solution space will look like at iteration 50?

(6) Is it possible to make the network "mechanically informed?" In other words, rather than just looking at elemental density values, can information about the stress, strain energy density, etc., be passed into the network to make more physics-based predictions? If so, how do the predictions of such networks compare to the predictions of the networks outlined in this thesis?

(7) How do these networks perform when applied to 3-dimensional problems?

(8) Can the networks be applied convolutionally to larger problems? To explain once more by example, consider a network which has been trained on 40x40 element regions. Can this network make "regional predictions" if applied convolutionally to a problem with a mesh size of 1600x1600 elements?

(9) Can these networks be applied to other computationally intensive problems of physical and engineering interest, such as turbulence or fluid-structure interactions?

(10) Can recurrent network models be used to solve the solution convergence problem? How do they compare to the performance of encoder-decoder networks when applied to this problem?

(11) Can these networks be used for problems where the solution space converges with abrupt or non-linear changes in material distribution?

(12) Can recurrent or sequential neural network models be used to solve this problem? What are the benefits and costs of using an encoder-decoder versus a sequential network?

Bibliography

[1] A Beginner's Guide to Generative Adversarial Networks (GANs). http://skymind.ai/wiki/generative-adversarial-network-gan. Accessed: 2019-04-01.

[2] History of Machine Learning. https://www.doc.ic.ac.uk/ jce317/history-machine-learning.html. Accessed: 2019-03-29.

[3] MNIST Demos on Yann LeCun's website. http://yann.lecun.com/exdb/lenet/. Accessed: 2019-04-05.

[4] Number of Possible Go Games at Sensei's Library. https://senseis.xmp.net/?NumberOfPossibleGoGames. Accessed: 2019-04-01.

[5] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. arXiv:1511.00561 [cs], November 2015. arXiv: 1511.00561.

[6] Martin P Bendsøe. Optimal shape design as a material distribution problem. Structural optimization, 1(4):193–202, 1989.

[7] Martin P Bendsøe and Ole Sigmund. Topology optimization by distribution of isotropic material. In Topology Optimization, pages 1–69. Springer, 2004.

[8] Martin Philip Bendsøe and Noboru Kikuchi. Generating optimal topologies in structural design using a homogenization method. Computer methods in applied mechanics and engineering, 71(2):197–224, 1988.

[9] Adit Deshpande. The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3). https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html. Accessed: 2019-04-06.

[10] Ross Girshick. Fast r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 1440–1448, 2015.

[11] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 580–587, 2014.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[14] William Hunter. Predominantly solid-void three-dimensional topology optimisation using open source software. PhD thesis, Stellenbosch: University of Stellenbosch, 2009.

[15] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In Advances in neural information processing systems, pages 2017–2025, 2015.

[16] Yoonsik Kim, Insung Hwang, and Nam Ik Cho. A New Convolutional Network-in-Network Structure and Its Applications in Skin Detection, Semantic Segmentation, and Artifact Reduction. arXiv:1701.06190 [cs], January 2017. arXiv: 1701.06190.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[18] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. page 10.

[19] Bernard Marr. A Short History of Machine Learning – Every Manager Should Read. `https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/`. Accessed: 2019-03-29.

[20] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015.

[21] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958.

[22] Sagar Sharma. What the Hell is Perceptron? `https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53`, September 2017.

[23] Thalles Silva. An intuitive introduction to Generative Adversarial Networks (GANs). `https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394`, January 2018. Accessed: 2019-04-01.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[25] Ivan Sosnovik and Ivan Oseledets. Neural networks for topology optimization. arXiv:1709.09578 [cs, math], September 2017. arXiv: 1709.09578.

[26] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[27] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 2818–2826, 2016.

[28] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–9, June 2015.

[29] Robail Yasrab, Naijie Gu, and Xiaoci Zhang. An encoder-decoder based convolution neural network (cnn) for future advanced driver assistance system (adas). Applied Sciences, 7(4):312, 2017.

[30] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In European conference on computer vision, pages 818–833. Springer, 2014.

[31] Ke Zhang, Miao Sun, Tony X Han, Xingfang Yuan, Liru Guo, and Tao Liu. Residual networks of residual networks: Multilevel residual networks. IEEE Transactions on Circuits and Systems for Video Technology, 28(6):1303–1314, 2018.