Fractals Created From Nonaffine Functions And Predicting Attractor Parameters Using Neural Networks

by
Liam Graham

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Mathematics & Statistics

Guelph, Ontario, Canada

ABSTRACT

FRACTALS CREATED FROM NONAFFINE FUNCTIONS AND PREDICTING ATTRACTOR PARAMETERS USING NEURAL NETWORKS

Liam Graham
University of Guelph, 2020

Advisor:
Dr. Matthew Demers

Fractals have a wide variety of applications in industries such as architecture and video games, as well as in research fields including soil mechanics and antenna design, amongst others. Finite collections of contractive set-valued affine functions, called iterated function systems (IFSs), have been shown to possess unique, globally attractive fixed points, termed attractors. These attractors often display fractal features. In this thesis we extend the theory of IFSs to include functions with bounded derivatives, and piecewise functions, both of which produce interesting results. Piecewise IFSs also allow for the development of fractal splicing. Additionally, we approximate a solution to a long-standing fractal inverse problem: Given the image of an attractor, what are the parameters of the IFS that produced it? We make use of neural networks to approximate this mapping.

Acknowledgements

This research was enabled in part by support provided by Compute Ontario, and Compute Canada. I am grateful for these resources and would not have been able to accomplish many of the things I did without them.

I would like to thank all the friends that I've made throughout this degree. From office birthday parties, to Math and Stats soccer and hockey teams, activities with them has made my time at Guelph very enjoyable.

Lastly, I am eternally grateful for everything my advisor, Matthew Demers, has done for me. The continual encouragement, time spent on personal, m ini-lectures, and guidance that he provided throughout this project have been invaluable, and are much appreciated.

Contents

Abstract **ii**

Acknowledgements **iii**

List of Tables **vi**

List of Figures **viii**

1 Introduction **1**

2 Fundamental Concepts in Iterated Function System Theory **5**
2.1 Mathematical Analysis Background 5
2.2 Iterated Function Systems 8
2.3 Generating Iterated Function System Attractors 18
2.4 Fractal Dimension 25

3 Creating Databases of Randomly Initialized Fractals **29**
3.1 Iterated Function Systems with Random Parameters 29
3.2 Properties of Randomly Generated Fractal Sets 35

4 Fractals Created From Nonaffine Functions **44**
4.1 Bounded Derivative Functions 45
4.2 Piecewise Functions 51

5 Deep Learning Background **66**
5.1 Feedforward Neural Networks 66
5.2 Activation Functions 71
5.3 Loss Functions 76
5.4 Gradient Descent and Optimization Algorithms 82
5.5 Convolutional Neural Networks 94
5.6 Applications To Hand-Written Digit Recognition 100
5.7 Regularization Methods 103
5.8 Convolutional Neural Network Architecture 112

6 Predicting Iterated Function System Parameters **122**
6.1 Designing The Model 123
6.2 Model Training, Analysis, And Performance 129

7 Future Work **138**

Bibliography **142**

A Supplementary Fractals **146**
A.1 Bounded Derivative Function Attractors 146
A.2 Bounded Range Function Attractors 146
A.3 Piecewise Affine Function Attractors 146
A.4 Model Produced IFS Attractors 146

B Source Code **155**
B.1 Generating Fractal Databases 155
B.2 Neural Network Implementation 182

List of Tables

2.1 Estimated fractal dimension for the Sierpinski triangle at various resolutions. 28

3.1 Effect of generating an attractor with more points on the pixels 36

5.1 XOR neural network model output compared to true values 74
5.2 Data used in Ex. 5.5 for linear regression by means of gradient descent. . . . 83
5.3 A comparison of all the networks discussed in Chapter 5.8 121

6.1 Calculating the accuracy of a neural network at various tolerance levels . . . 127
6.2 The learning rates for each neural network model in Chapter 6 128
6.3 Hyperparameter changes throughout training of each model in Chapter 6 . . 132
6.4 The accuracy of each model at various tolerance levels 134

List of Figures

2.1 A visualization of the Hausdorff distance 11
2.2 A visual depiction of the sets used in Ex. 2.2 15
2.3 The fixed point of an IFS that resembles as tree. 17
2.4 The fixed point of an IFS that looks like a maple leaf. 18
2.5 Example of the first nine iterations of the deterministic algorithm 21
2.6 Generating the Barnsley fern with the random algorithm 24
2.7 Calculating the fractal dimension of the Sierpinski triangle by segmentation . 27

3.1 Attractor sets pertaining to IFSs with randomly initialized parameters . . . 32
3.2 The attractor corresponding to the randomly generated IFS in Ex. 3.2 34
3.3 Comparing absolute multiplicative parameter distributions of sparse fractals 39
3.4 Comparing absolute multiplicative parameter distributions of dense fractals . 40
3.5 Distribution of the number of pixels of a fractal for each database 42
3.6 Distribution of the standard deviation of fractal pixels for each database . . 43

4.1 Attractors from bounded derivative functions 47
4.2 Attractors from functions with a derivative bound of one 49
4.3 Bounded derivative function extensions of the Levy C fractal 50
4.4 Attractors from piecewise-affine IFSs with single component boundaries . . . 54
4.5 Attractors from piecewise-affine IFSs 55
4.6 Iterations of the deterministic algorithm for an piecewise-affine IFS 57
4.7 How overlapping piecewise function conditions can be made disjoint 58
4.8 A simplistic depiction of the components of a piecewise-affine attractor . . . 59
4.9 Randomly initialized piecewise-affine IFSs with square and circle boundaries 61
4.10 Examples of fractal splicing with some common affine IFSs. 63
4.11 Effects of incorporating a new mapping to an IFS for fractal splicing 64
4.12 The effects of the piecewise boundary on fractal splicing 65

5.1 A graphical representation of the simple neural network in Ex. 5.1 68
5.2 An example of a neural network that contains bias nodes. 70
5.3 A plot of the sigmoid function as well as its derivative. 73
5.4 The neural network used to model the XOR function in Ex. 5.2 74
5.5 Results of using gradient descent for linear regression in Ex. 5.5. 84
5.6 A visualization of how the gradient of a neural network is calculated 86

5.7 Results of using stochastic gradient descent for linear regression in Ex. 5.7. . 89
5.8 A visual depiction of the cross correlation operation in CNNs 95
5.9 A visual depiction of a filter that acts on multiple feature maps 98
5.10 A visual depiction of the model used to classify hand written digits 102
5.11 A diagram of the last two layers of a network used in Ex. 5.11 110
5.12 The structural design of AlexNet, the 2012 ILSVRC winner. 113
5.13 The architecture of each of the VGGNet variations used in the 2014 ILSVRC 115
5.14 The inception module, or block used in GoogLeNet. 118
5.15 The architecture of GoogLeNet, the 2014 ILSVRC winner 118
5.16 The design of each of the ResNet models used in the 2015 ILSVRC 120

6.1 The block structure used in our model to predict IFS parameters 124
6.2 The loss and validation accuracy of the two-function IFS model 130
6.3 The loss and validation accuracy of each model throughout training. 131
6.4 Distribution of the additive parameters of an IFS 133
6.5 Attractors of model produced IFSs in comparison with the model input . . . 135
6.6 Lower sparsity attractors of model produced IFSs in comparison to the input 136
6.7 Low sparsity attractors of the two-function IFS model with alternative inputs 137

7.1 Attractors from bounded range functions 141

A.1 Derivative bounded function IFS attractors such that $g_1 = g_3 = cos(b_i x)/b_i$. . 147
A.2 Derivative bounded function IFS attractors such that $g_1 = g_3 = sin(b_i x)/b_i$. . 148
A.3 Derivative bounded function IFS attractors such that $g_1 = g_3 = tanh(b_i x)/b_i$. 149
A.4 Range bounded function IFS attractors such that $g_1 = g_3 = cos(b_i x)$. 150
A.5 Range bounded function IFS attractors such that $g_1 = g_3 = sin(b_i x)$ 151
A.6 Range bounded function IFS attractors such that $g_1 = g_3 = tanh(b_i x)$. 152
A.7 Additional piecewise-affine IFS attractors with various boundaries 153
A.8 Low sparsity model produced IFS attractors in comparison with the input . 154

Chapter 1

Introduction

Fractals can be thought of as infinite sets that possess self-similar features. These objects, in part known for their ability to resemble nature quite well, have a wide variety of applications. While they have been used in areas such as architecture, video games, movies, and computer graphics for their appealing physical manifestations, they have also been useful in the fields of soil mechanics [1], antenna design [2], and many others. One possible driving force behind this array of applicable fields is their fractional dimension property. That is, we can define a dimension for these infinite sets that is strictly non-integer, unlike the three dimensional world we live in. Possessing this fractional dimension is another possible definition for a fractal set. So how exactly can we create one?

The principal mechanism that allows us to create fractals lies behind Banach's fixed point theorem: A contraction mapping on a complete metric space possesses a unique, globally attractive fixed point. Thus, repeated iteration of a contractive function on a point in a complete metric space will always lead to the same fixed point for that function, no matter the starting location. Combining this idea with iterated function systems (IFSs); a finite collection of contractive set-valued functions, is one method of creating sets possessing these fractal properties. In other words, these IFSs satisfy Banach's fixed point theorem and have unique, globally attractive fixed points themselves, known as attractors, which are infinite sets, and often contain self-similar features and a fractional dimension.

Of course, since fractal sets are infinite, we can only ever obtain approximations of them. Once we have an IFS satisfying the proper contractivity conditions, we can apply a specialized version of fixed point iteration, often referred to as the chaos game, to generate points approximating its attractor. This algorithm begins with a point and randomly selects a function from the IFS to apply. Functions from the IFS then continue to be randomly selected and applied to the output of the previous function. The resulting sequence of outputs from each transformation quickly converges to the attractor and accurately portrays it when graphed.

Almost all work done on fractals created from IFSs explores solely IFSs consisting of affine maps. We will introduce new functions from which we can build an IFS and generate fractal approximations. Namely, functions that have a bounded derivative, and piecewise functions. We find that both types of IFSs produce interesting features in their attractors. The functions we explore that have a bounded derivative result in attractors that still maintain many features of affine functions. For IFSs constructed from piecewise functions, we only examined those for which each branch of the piecewise function is affine. Though, these functions are still quite interesting as they manifest the boundary between the functions in their attractors. For both types of alternative IFSs, we are able to produce extensions of common affine-consisting IFSs. In terms of the piecewise functions, this takes the form of fractal splicing: Combining the attractors of two IFSs by means of piecewise functions.

As a result of the many applications of fractals, there has been a long-standing inverse problem in the field of fractal geometry: Given an image of a fractal, what are the parameters of the IFS that generated it? There have been several approaches to this problem already, such as moment based methods [3, 4], making use of wavelet transforms [5], evolutionary algorithms [6], and most recently, a swarm intelligence method called the cuckoo search [7]. These previous approaches give satisfactory results for several fractals, however, they fail to solve the problem in a more general setting. In this thesis, we attempt to solve this problem not by doing an exhaustive search for each attractor, but by obtaining a function that maps the image of an attractor to IFS parameter values. In order to perform such a task, we will approximate the function with a neural network.

Neural networks are nested vector-valued functions that optimize their parameters through gradient descent to approximate other functions. There have been many recent advances in this field allowing for technologies such as speech recognition and prediction [8], algorithms that can outperform humans at games such as chess and go [9], and many others. Convolutional neural networks are a specialized type of function approximator for use in image-related applications. These networks are the key mechanism behind self-driving cars, and object detection and labelling algorithms, amongst others [8]. We will use these convolutional neural networks in order to map a given attractor to its IFS parameter values.

In order to train the network to perform the desired task, we will use a method called supervised learning. This type of machine learning has the model compute an answer which then gets compared with known values in order to calculate an error, and update parameters of the network. Supervised learning often requires vast amounts of data, hence, we will create large databases of randomly initialized approximated fractal9 sets in order to accomplish this task.

The second chapter of this work covers the necessary tools to understand iterated function system theory, much of which can be found from [10]. It defines the required mathematical concepts and theorems, and describes algorithms to approximate IFS attractors, as well as how one can calculate the fractal dimension.

Chapter 3 is novel work that uses the tools of Chapter 2 to construct conditions that must be met in order to produce an IFS guaranteed to possess an attractor when all parameters are randomly generated. We then use these conditions to generate fractal databases and analyse some of their properties.

In Chapter 4 we present original work that develops theory for the previously mentioned nonaffine functions to be used in IFSs. We extend theorems from Chapter 2 in order to show their convergence to an attractor, and provide examples of these fractals. We also provide methods of extending common IFSs to nonaffine IFSs.

The fundamental ideas behind neural networks are covered in Chapter 5. This chapter covers the basic framework of this machine learning technique, as well as its implementation, common practices, and optimization methods. Many of the concepts in this chapter stem from [8]. We also cover network designs of competition winning models in this field from which we will derive principles used in our own network structure.

The creation, training, and results of the neural network used to predict IFS parameter values is given in Chapter 6. Here we justify the design choices of our model, provide metrics of its performance, and reconstruct attractors from the model output for comparison with the original.

Future work of both nonaffine extensions of IFSs, as well as the neural network implementation can subsequently be found in Chapter 7. Some supplementary examples of nonaffine function IFS attractors as well as source code for this work can be found in the appendix.

Chapter 2

Fundamental Concepts in Iterated Function System Theory

There are many different kinds of fractals as well as ways to generate them. In this chapter we define and derive the tools necessary to understand fractals generated from iterated function systems. Additionally, we cover methods of generating them, and methods for calculating the fractal dimension.

2.1 Mathematical Analysis Background

The theory behind fractals lies heavily in mathematical analysis. We will build these tools from some familiar definitions that can be found in [10].

Definition 2.1 (Metric Space)**.** *A metric space is the pairing of a space, X, with a metric, or distance measuring function, $d : X \times X \to \mathbb{R}$, where d satisfies the following properties:*

1.	$d(x,y) = d(y,x)$	$\forall x, y \in X$
2.	$0 < d(x,y) < \infty$	$\forall x, y \in X, x \neq y$
3.	$d(x,x) = 0$	$\forall x \in X$
4.	$d(x,z) \leq d(x,y) + d(y,z)$	$\forall x, y, z \in X$

Definition 2.2 (Sequence of Bounded Variation)**.** *Let $\{x_n\}_{n=1}^{\infty}$ be a sequence in a metric space (X,d). Then it is called a sequence of bounded variation if $\sum_{n=1}^{\infty} d(x_{n+1}, x_n) < \infty$.*

Definition 2.3 (Cauchy Sequence)**.** *Let $\{x_n\}_{n=1}^{\infty}$ be a sequence in a metric space (X,d). Then it is called a Cauchy sequence if for any $\epsilon > 0$, $\exists\, N_\epsilon \in \mathbb{N}$ such that if m, $n \in \mathbb{N}$ and m, $n \geq N_\epsilon$, then $d(x_m, x_n) < \epsilon$.*

Definition 2.4 (Complete Metric Space)**.** *Let (X,d) be a metric space. Then the following are equivalent:*

- *(X,d) is a complete metric space*
- *Every sequence of bounded variation in X converges to a point in X*
- *Every Cauchy sequence in X converges to a point in X*

Definition 2.5 (Contractive Function)**.** *A function $f : X \to X$ is called contractive on a given metric space (X,d) if $\exists\, c \in [0,1)$ such that $d(f(x), f(y)) \leq c \cdot d(x,y)$, $\forall x, y \in X$. Then f is called a contraction mapping, and c, the contraction factor.*

Definition 2.6 (Fixed Point)**.** *Let (X,d) be a metric space, and $f : X \to X$. f possesses a fixed point, $\bar{x} \in X$, if $f(\bar{x}) = \bar{x}$.*

Definition 2.7 (Open Cover and Subcover)**.** *Let S be a subset of a metric space $(X,\, d)$, and $\{U_i\}_{i \in I}$ be a collection of open sets which could be finite, or infinite. Then if $S \subseteq \bigcup_{i \in I} U_i$, $\{U_i\}_{i \in I}$ is called an open cover. Additionally, if $\{U_i\}_{i \in J}$ where $J \subset I$ is an open cover for S, then $\{U_i\}_{i \in J}$ is called a subcover for S.*

Definition 2.8 (Compact Set)**.** *Let S be a subset of a metric space (X, d). Then S is said to be compact if every open cover of S has a finite subcover, that is, a subcover containing a collection of a finite number of open sets.*

These definitions are necessary as the type of fractals that will be used throughout this thesis are created from contractive functions that act on compact subsets of complete metric spaces. They also rely on the following theorem:

Theorem 2.1 (Banach's Fixed Point Theorem)**.** *Let (X, d) be a complete metric space. Then if $f : X \to X$ is a contraction mapping, it possesses a unique, globally attractive fixed point.*

Proof. Let (X, d) be a complete metric space and $f : X \to X$ be a contractive function, that is, $\exists\, c \in [0, 1)$ such that $d(f(x), f(y)) \leq c \cdot d(x, y)$, $\forall x, y \in X$. Now, let $x_0 \in X$ and $x_{n+1} = f(x_n)$, $\forall n \in \mathbb{N}$. We begin by noting that

$$\begin{aligned} d(x_n, x_{n+1}) &= d(f(x_{n-1}), f(x_n)) \\ &\leq c \cdot d(x_{n-1}, x_n). \end{aligned}$$

By induction we see that

$$d(x_n, x_{n+1}) \leq c^n \cdot d(x_0, x_1).$$

Hence the series of distances between points is given by

$$\sum_{n=0}^{\infty} d(x_n, x_{n+1}) \leq \sum_{n=0}^{\infty} c^n \cdot d(x_0, x_1) = \frac{1}{1-c} d(x_0, x_1),$$

which is clearly finite. Thus $\{x_n\}_{n=0}^{\infty}$ is a sequence of bounded variation and therefore converges in a complete metric space. Let $\bar{x} = \lim_{n\to\infty} x_n$. Then $\bar{x}$ is a fixed point of f, hence $f(\bar{x}) = \bar{x}$. Suppose now that f has a second fixed point, $\bar{x}'$. Then,

$$\begin{aligned} d(\bar{x}, \bar{x}') = d(f(\bar{x}), f(\bar{x}')) &\leq c \cdot d(\bar{x}, \bar{x}') \\ 0 &\geq (1-c)d(\bar{x}, \bar{x}'). \end{aligned}$$

This is only possible if $\bar{x} = \bar{x}'$ since $c \in [0,\, 1)$ making $(1-c)$ positive, and a valid metric must satisfy $d(x,\, y) \geq 0$ and $d(x,\, y) = 0 \implies x = y$. Thus, f has a unique fixed point in X. Additionally, we can see this fixed point is globally attractive as we can start the sequence $\{x_n\}_{n=0}^{\infty}$ with any $x_0 \in X$ and use the iteration shown in this proof to obtain $\bar{x}$. □

Now we have all the mathematical tools needed to talk about fractals, we can begin examining iterated function systems; the means by which we will generate fractals throughout this thesis.

2.2 Iterated Function Systems

A simple method of generating a fractal set is through an iterated function system (IFS)[10]. An IFS is essentially a finite collection of contractive functions defined on a complete metric space, and can be formally defined as follows:

Definition 2.9 (Iterated Function System)**.** *Let $(X,\, d)$ be a complete metric space and $W = \{\hat{w}_i : X \to X\}_{n=1}^{N}$ be a set of contractive functions. Then W is an IFS, or sometimes called N-map IFS.*

An IFS alone does not generate a fractal, though. We require a means through which we can apply an IFS to points in a metric space. These means are provided by the Hutchinson operator [11].

Definition 2.10 (Hutchinson operator)**.** *Let $W = \{\hat{w}_i : X \to X\}_{n=1}^{N}$ be an IFS acting on the space $(X,\, d)$, and $S \subseteq X$ be non-empty and compact. Then the Hutchinson operator is given by*

$$H_W(S) = \bigcup_{i=1}^{N} \hat{w}_i(S),$$

where

$$\hat{w}_i(S) = \{w_i(x) \,|\, x \,\in\, S\}.$$

As we can see, the Hutchinson operator acts on sets of points in X, rather than a single point, though it could of course be applied to any singleton set. Given a complete metric space X, we can generate another metric space that we will denote $\mathcal{H}(X)$; the space composed of all compact, non-empty subsets of X. We wish to show that the Hutchinson operator is itself a contractive mapping in this space. In order to do so, we must find a metric for $\mathcal{H}(X)$ that measures distances between sets, and we would particularly like one that makes it a complete space. We begin by defining the distance from a point to a set.

Definition 2.11 (Point-Set Distance)**.** *Let $(X,\, d_X)$ be a complete metric space, $x \,\in\, X$, and $S \,\subseteq\, X$. Then the point-set distance is given by,*

$$d_{X,\,point}(x,\, S) = \inf_{y\,\in\,S} d_X(x,\, y).$$

Note that the right-hand side of the equation is well defined as X is a complete space.

We can now define the distance between two sets in a similar manner.

Definition 2.12 (Set-Set Distance)**.** *Let $(X,\, d_X)$ be a complete metric space, $R \subseteq X$ and,*

$S \subseteq X$. *Then the set-set distance is given by,*

$$\begin{aligned} d_{X,\,set}(R,\,S) &= \sup_{x\,\in\,R} d_{X,\,point}(x, S) \\ &= \sup_{x\,\in\,R}\left(\inf_{y\,\in\,S} d_X(x, y)\right). \end{aligned}$$

In words, we can calculate $d_{X,\,set}(R,\,S)$ as follows: Take a point in R and find the greatest lower bound of its distance to S. Then, do this to all points in R to obtain a set of greatest lower bounds. $d_{X,\,set}(R,\,S)$ is then the least upper bound of that set. An example may help demonstrate what is happening.

Example 2.1. *Working in* $\mathbb{R}$*, let* $R = [0,\,1]$*,* $S = [2,\,4]$*, and take the distance function to be the natural metric for* $\mathbb{R}$*;* $d_{\mathbb{R}}(x,\,y) = |x - y|$*. Calculating the distance both ways we see,*

$$\begin{aligned} d_{\mathbb{R},\,set}(R,\,S) &= \sup_{x\,\in\,R}\left(\inf_{y\,\in\,S} d_{\mathbb{R}}(x, y)\right) & \qquad d_{\mathbb{R},\,set}(S,\,R) &= \sup_{x\,\in\,S}\left(\inf_{y\,\in\,R} d_{\mathbb{R}}(x, y)\right) \\ &= \sup_{x\,\in\,R} d_{\mathbb{R}}(x,\,2) & &= \sup_{x\,\in\,S} d_{\mathbb{R}}(x,\,1) \\ &= d_{\mathbb{R}}(0,\,2) & &= d_{\mathbb{R}}(4,\,1) \\ &= 2 & &= 3. \end{aligned}$$

As can be seen from the above example, this distance function does not satisfy the symmetry property (the first requirement in Def. 2.1), and therefore is not a valid metric. Additionally, $d_{X,\,set}(R,\,S) = 0$ does not imply equality; both of these problems can be fixed with a small modification. One may also note that $d_{X,\,set}(R,\,S)$ may be infinite for sets that are not compact. This problem will not matter as long as we only pair it with $\mathcal{H}(X)$.

Definition 2.13 (Hausdorff Distance)**.** *Let* $(X,\,d_X)$ *be a metric space,* $R \subseteq X$*, and* $S \subseteq X$*.*

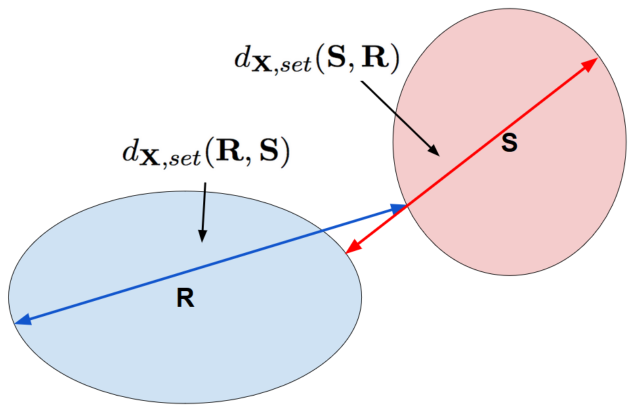

$d_{\mathbf{X},set}(\mathbf{S},\mathbf{R})$
$d_{\mathbf{X},set}(\mathbf{R},\mathbf{S})$
S
R

Figure 2.1: A visualization of the Hausdorff distance. The Hausdorff distance is given by the larger of the two distances shown.

Then the Hausdorff distance is given by

$$d_{\mathcal{H}}(R,\, S) = \max\{d_{X,\,set}(R,\, S),\, d_{X,\,set}(S,\, R)\}.$$

This can be visualized my taking the maximum of the two distances shown in Fig. 2.1.

It can easily be seen that the Hausdorff distance is a valid metric. In addition, it happens to form a complete metric space with $\mathcal{H}(X)$.

Theorem 2.2. *Let $(X,\, d)$ be a complete metric space. Then its induced Hausdorff metric space, $(\mathcal{H}(X),\, d_{\mathcal{H}})$, is also complete.*

Proof. Here we provide only a sketch of the proof as the full proof is quite long and complex. For more, see [10, 12]. Begin by constructing a Cauchy sequence $\{A_n\}$ in $(\mathcal{H}(X),\, d_{\mathcal{H}})$. Then, let A be the set of all points $x \in X$ such that there is a sequence $\{x_n\}$ converging to it, where $x_n \in A_n \forall n$. The next step is to show that $A \in \mathcal{H}(X)$, which can be done using the fact that $\{A_n\}$ is Cauchy and showing that $A \subseteq A_n + \epsilon$, where for a set S, $S + \epsilon$ is the set $\{x \in X : d_{X,\,point}(x,\, S) \leq \epsilon\}$. Lastly, it can be shown that $A_n \subseteq A + \epsilon$ and hence $d_{\mathcal{H}}(A_n,\, A) \leq \epsilon$, thus $\{A_n\}$ converges to A, and $(\mathcal{H}(X),\, d_{\mathcal{H}})$ is complete. □

With the pairing of $\mathcal{H}(X)$ and the Hausdorff distance being a complete space, if we can show that the Hutchinson operator is a contraction mapping in this space for any given IFS, then we can apply Banach's fixed point theorem. In order to do this, we will require Lemma 2.3, which can be found in [10].

Lemma 2.3. *Let $(X,\, d)$ be a complete metric space. Then for any sets $S_1,\, S_2,\, S_3,\, S_4 \in \mathcal{H}(X)$, we have that*

$$d_{\mathcal{H}}(S_1 \cup S_2,\, S_3 \cup S_4) \leq \max\{d_{\mathcal{H}}(S_1,\, S_3),\, d_{\mathcal{H}}(S_2,\, S_4)\}.$$

Proof. Let $S_1,\, S_2,\, S_3,\, S_4 \in \mathcal{H}(X)$. Then we see that

$$\begin{aligned} d_{X,\,set}(S_1 \cup S_2,\, S_3 \cup S_4) &= \sup_{x \in S_1 \cup S_2} d_{X,\,point}(x,\, S_3 \cup S_4) \\ &= \max\left\{ \sup_{x\in S_1} d_{X,\,point}(x,\, S_3 \cup S_4),\, \sup_{x \in S_2} d_{X,\,point}(x,\, S_3 \cup S_4) \right\} \\ &= \max\{d_{X,\,set}(S_1,\, S_3 \cup S_4),\, d_{X,\,set}(S_2,\, S_3 \cup S_4)\}. \end{aligned}$$

Similarly, we see

$$\begin{aligned} d_{X,\,set}(S_1,\, S_3 \cup S_4) &= \sup_{x \in S_1} \left(\inf_{y \in S_3 \cup S_4} d(x,\, y) \right) \\ &= \sup_{x \in S_1} \left(\min\left\{ \inf_{y \in S_3} d(x,\, y),\, \inf_{y \in S_4} d(x,\, y) \right\} \right) \\ &\leq \min\left\{ \sup_{x \in S_1} \left(\inf_{y\in S_3} d(x,\, y) \right),\, \sup_{x \in S_1} \left(\inf_{y \in S_4} d(x,\, y) \right) \right\} \\ &= \min\{d_{X,\,set}(S_1,\, S_3),\, d_{X,\,set}(S_1,\, S_4)\}. \end{aligned}$$

With the minimum in this result, we can choose $d_{X,\,set}(S_1,\, S_3 \cup S_4) \leq d_{X,\,set}(S_1,\, S_3)$ and $d_{X,\,set}(S_2,\, S_3 \cup S_4) \leq d_{X,\,set}(S_2,\, S_4)$. It follows that $d_{X,\,set}(S_3,\, S_1 \cup S_2) \leq d_{X,\,set}(S_3,\, S_1)$ and

$d_{X,\,set}(S_4,\, S_1 \cup S_2) \leq d_{X,\,set}(S_4,\, S_2)$. Hence we get the following:

$$\begin{aligned}
d_{\mathcal{H}}(S_1 \cup S_2,\, S_3 \cup S_4) &= \max\{d_{X,\,set}(S_1 \cup S_2,\, S_3 \cup S_4),\, d_{X,\,set}(S_3 \cup S_4,\, S_1 \cup S_2)\} \\
&\leq \max\Big\{ \max\{d_{X,\,set}(S_1,\, S_3),\, d_{X,\,set}(S_2,\, S_4)\}, \\
&\qquad\qquad \max\{d_{X,\,set}(S_3,\, S_1),\, d_{X,\,set}(S_4,\, S_2)\}\Big\} \\
&= \max\Big\{ \max\{d_{X,\,set}(S_1,\, S_3),\, d_{X,\,set}(S_3,\, S_1)\}, \\
&\qquad\qquad \max\{d_{X,\,set}(S_2,\, S_4),\, d_{X,\,set}(S_4,\, S_2)\}\Big\} \\
&= \max\{d_{\mathcal{H}}(S_1,\, S_3),\, d_{\mathcal{H}}(S_2,\, S_4)\}.
\end{aligned}$$

□

Before continuing to the proof of the Hutchinson operator being a contraction mapping, chapter four will require knowing that equality can be achieved in this lemma when S_1 is disjoint from S_2, and S_3 is disjoint from S_4. This can be shown by first noting that in the above choice to set $d_{X,\,set}(S_1,\, S_3 \cup S_4) \leq d_{X,\,set}(S_1,\, S_3)$, we could have instead chosen $d_{X,\,set}(S_1,\, S_3 \cup S_4) \leq d_{X,\,set}(S_1,\, S_4)$. This would lead to a different inequality of the form

$$d_{\mathcal{H}}(S_1 \cup S_2,\, S_3 \cup S_4) \leq \max\{d_{\mathcal{H}}(S_1,\, S_4),\, d_{\mathcal{H}}(S_2,\, S_3)\},$$

which is equivalent to switching the labels of the sets in one of the unions. With this label switching, we can assume S_3 to always be the closer set to S_1 out of S_3 and S_4, and use the original statement of the lemma. Then the only relation in the proof preventing equality is

$$\begin{aligned}
\sup_{x \in S_1} \left(\min\Big\{ \inf_{y \in S_3} d(x,\, y),\, \inf_{y \in S_4} d(x,\, y) \Big\} \right) &\leq \min\left\{ \sup_{x \in S_1} \Big(\inf_{y \in S_3} d(x,\, y) \Big),\, \sup_{x \in S_1} \Big(\inf_{y \in S_4} d(x,\, y) \Big) \right\} \\
&= \sup_{x \in S_1} \Big(\inf_{y \in S_3} d(x,\, y) \Big).
\end{aligned}$$

We can now see that this will be an equality for disjoint compact subsets of $\mathbb{R}^2$.

Example 2.2. *Working in $\mathbb{R}^2$ with the Euclidean metric, let the sets S_1, S_2, S_3, $S_4 \in \mathcal{H}(\mathbb{R}^2)$ be defined as in Fig. 2.2. Following the process outlined above, we first calculate the left-hand side of the Lemma. The largest value in the set of minimum distances from points in $S_1 \cup S_2$ to points in $S_3 \cup S_4$ would be*

$$\begin{aligned} d_{\mathbb{R}^2,\,set}(S_1 \cup S_2,\, S_3 \cup S_4) &= \sup_{(x_1,\,y_1)\,\in\, S_1 \cup S_2} \left(\inf_{(x_2,\,y_2)\,\in\, S_3 \cup S_4} d_{\mathbb{R}^2}((x_1,\, y_1),\, (x_2,\, y_2)) \right) \\ &= d_{\mathbb{R}^2}((-3,\, 3),\, (-3,\, -1)) \\ &= 4 \end{aligned}$$

Similarly, the set-set distance calculated the other way yields,

$$\begin{aligned} d_{\mathbb{R}^2,\,set}(S_3 \cup S_4,\, S_1 \cup S_2) &= \sup_{(x_1,\,y_1)\,\in\, S_3 \cup S_4} \left(\inf_{(x_2,\,y_2)\,\in\, S_1 \cup S_2} d_{\mathbb{R}^2}((x_1,\, y_1),\, (x_2,\, y_2)) \right) \\ &= d_{\mathbb{R}^2}((-3,\, -3),\, (-3,\, 1)) \\ &= 4 \end{aligned}$$

Then the Hausdorff distance is $\max(4,\, 4) = 4$*. All points used to calculate both these distances are either in S_1 or S_3, hence it is clear that Lemma 2.3 holds for this example.*

Theorem 2.4. *Let $W = \{\hat{w}_i : X \to X\}_{n=1}^{N}$ be an IFS on a complete metric space $(X,\, d)$. Then the Hutchinson operator as defined in Def. 2.10 is a contraction mapping in $(\mathcal{H}(X),\, d_{\mathcal{H}})$.*

Proof. We will prove this theorem only for 2-map IFSs, however, it can easily be extended by induction to IFSs with N contraction mappings. Let R, $S \in \mathcal{H}(X)$. Then applying

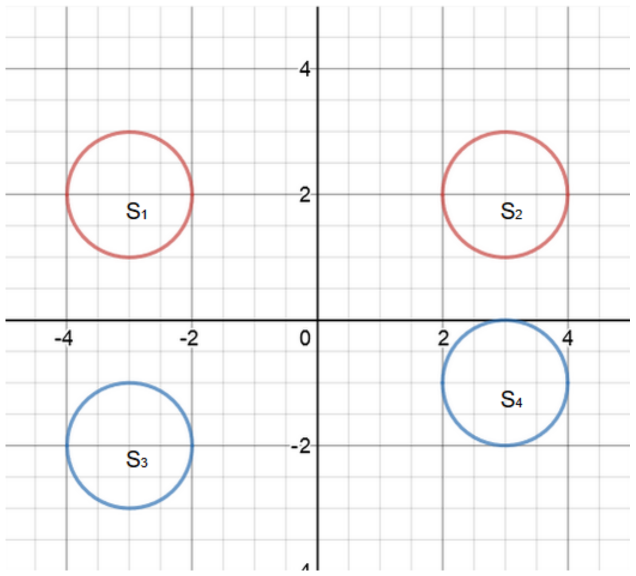


Figure 2.2: A visual depiction of the sets used in Ex. 2.2

Lemma 2.3 to these sets results in

$$\begin{aligned} d_{\mathcal{H}}(H_W(R),\, H_W(S)) &= d_{\mathcal{H}}(\hat{w}_1(R) \cup \hat{w}_2(R),\, \hat{w}_1(S) \cup \hat{w}_2(S)) \\ &\leq \max\{d_{\mathcal{H}}(\hat{w}_1(R),\, \hat{w}_1(S)),\, d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\} \\ &\leq \max\{c_1,\, c_2\} d_{\mathcal{H}}(R,\, S). \end{aligned}$$

where c_1 and c_2 are the contraction factors of the $\hat{w}_1$ and $\hat{w}_2$ mappings respectively. Thus H_W is a contraction mapping on $(\mathcal{H}(X),\, d_{\mathcal{H}})$. □

We have now satisfied all the requirements to apply Banach's fixed point theorem, hence the Hutchinson operator has a unique, globally attractive fixed point in $(\mathcal{H}(X),\, d_{\mathcal{H}})$.

Definition 2.14 (Attractor)**.** *Let W be an IFS. Then W has a fixed point in $(\mathcal{H}(X),\, d_{\mathcal{H}})$ called the attractor, given by*

$$A = H_W(A) = \bigcup_{i=1}^{N} \hat{w}_i(A).$$

We can find the attractor of a given IFS by using the iteration given in the proof of Banach's fixed point theorem. If (X, d) is the metric space on which the functions in an IFS are defined, then for any compact subset $S_0 \subseteq X$, we can define $S_{n+1} = H_W(S_n)$. The attractor, A, is found by taking the limit $\lim_{n\to\infty} S_n = A$. This fixed point iteration often leads to fractal properties; namely infinitesimal details and self similarity in the sense that the attractor is comprised of shrunken, distorted copies of itself.

Example 2.3. *In this example we show how some fractals can resemble nature quite well. For example, the attractor produced from the IFS*

$$W = \begin{cases} \hat{w}_1(x, y) &= \begin{bmatrix} 0.195 & -0.488 \\ 0.344 & 0.443 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.4431 \\ 0.2452 \end{bmatrix} \\ \hat{w}_2(x, y) &= \begin{bmatrix} 0.462 & 0.414 \\ -0.252 & 0.361 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.2511 \\ 0.5692 \end{bmatrix} \\ \hat{w}_3(x, y) &= \begin{bmatrix} -0.058 & -0.070 \\ 0.452 & -0.111 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.5976 \\ 0.0969 \end{bmatrix} \\ \hat{w}_4(x, y) &= \begin{bmatrix} -0.035 & 0.070 \\ -0.469 & -0.022 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.4884 \\ 0.5069 \end{bmatrix} \\ \hat{w}_5(x, y) &= \begin{bmatrix} -0.637 & 0.000 \\ 0.000 & 0.501 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.8562 \\ 0.2513 \end{bmatrix} \end{cases}$$

is shown in Fig. 2.3 and resembles a tree. There are many objects in nature which we can

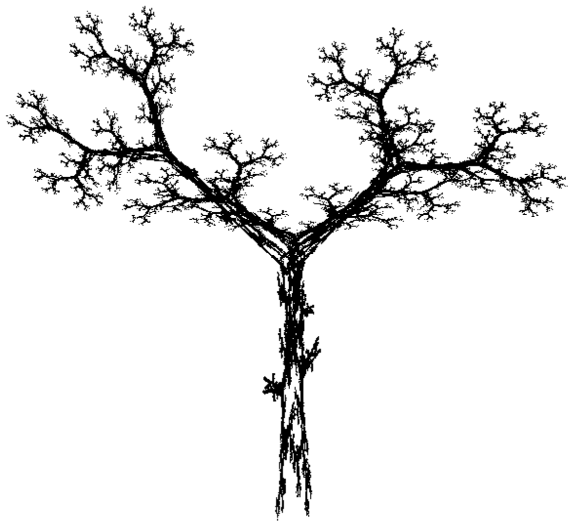


Figure 2.3: The fixed point of an IFS that resembles as tree.

simulate with a fractal. The maple leaf shown in Fig. 2.4 was made with the IFS

$$W = \begin{cases} \hat{w}_1(x, y) &= \begin{bmatrix} 0.14 & 0.01 \\ 0.00 & 0.51 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.08 \\ -1.31 \end{bmatrix} \\ \hat{w}_2(x, y) &= \begin{bmatrix} 0.43 & 0.52 \\ -0.45 & 0.50 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1.49 \\ -0.75 \end{bmatrix} \\ \hat{w}_3(x, y) &= \begin{bmatrix} 0.45 & -0.49 \\ 0.47 & 0.47 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -1.62 \\ -0.74 \end{bmatrix} \\ \hat{w}_4(x, y) &= \begin{bmatrix} 0.49 & 0.00 \\ 0.00 & 0.51 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.02 \\ 1.62 \end{bmatrix} \end{cases}$$

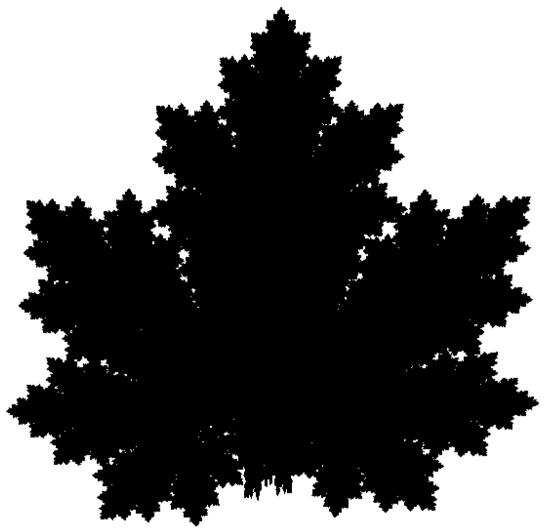


Figure 2.4: The fixed point of an IFS that looks like a maple leaf.

2.3 Generating Iterated Function System Attractors

Attractors can have very interesting features. As we just saw in the previous section, some of them resemble objects we see in nature such as plants and trees. So how exactly do we obtain a picture of an attractor? The true answer is that we cannot. Attractors contain infinitesimal details that are impossible to completely capture graphically, and therefore we can only ever obtain an approximation. To obtain an approximation though, we have already briefly talked about one method. Both methods found in this section can be found in [10].

Definition 2.15 (Deterministic Algorithm)**.** *Let $(X,\, d)$ be a complete metric space, and $W = \{\hat{w}_i\}_{i=1}^{N}$ be an IFS consisting of contractive functions on X. Then the deterministic algorithm goes as follows: Pick an initial compact subset of X, the easiest would be a point $S_0 = \{x_0\}$. Then, perform fixed point iteration by repeatedly applying the Hutchinson operator. The first* *iteration will result in the set*

$$\begin{aligned} S_1 &= H_W(S_0) \\ &= \bigcup_{i=1}^{N} \hat{w}_i(x_0) \\ &= \{\hat{w}_1(x_0), \, \ldots \, , \, \hat{w}_N(x_0)\}. \end{aligned}$$

Applying the operator again yields

$$\begin{aligned} S_2 = &\bigcup_{i,\,j=1}^{N} \hat{w}_i(\hat{w}_j(x_0)) \\ =&\{\hat{w}_1(\hat{w}_1(x_0)), \, \ldots \, , \, \hat{w}_1(\hat{w}_N(x_0), \, \hat{w}_2(\hat{w}_1(x_0)), \\ &\ldots \, , \, \hat{w}_2(\hat{w}_N(x_0)), \, \ldots \, , \, \hat{w}_N(\hat{w}_N(x_0))\} \end{aligned}$$

The more iterations of the deterministic algorithm we complete, the more points we have, and the better the approximation of the attractor. We can then plot the points to visualize the attractor of a given IFS.

Example 2.4. *In this example we use the unit square in $\mathbb{R}^2$ as our initial set with the* *following IFS:*

$$W = \begin{cases} \hat{w}_1(x,\, y) &= \begin{bmatrix} 0.255 & 0 \\ 0 & 0.255 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.3726 \\ 0.6714 \end{bmatrix} \\ \hat{w}_2(x,\, y) &= \begin{bmatrix} 0.255 & 0 \\ 0 & 0.255 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.1146 \\ 0.2232 \end{bmatrix} \\ \hat{w}_3(x,\, y) &= \begin{bmatrix} 0.255 & 0 \\ 0 & 0.255 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.6306 \\ 0.2232 \end{bmatrix} \\ \hat{w}_4(x,\, y) &= \begin{bmatrix} 0.37 & -0.642 \\ 0.642 & 0.37 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.6356 \\ -0.0061 \end{bmatrix} \end{cases}$$

Plotting the first 9 *iterations of the deterministic algorithm, we get the images in Fig. 2.5.*

The problem with the deterministic algorithm is that the number of points grows much too quickly. If we begin with m points in our initial set, then after n iterations of this algorithm we will have mN^n points in S_n. So, for example, with $m = 1$, an IFS consisting of $N = 4$ functions, and we perform $n = 10$ iterations, we already have over a million points! This algorithm tends to be very computationally expensive as there are larger and larger gaps in the number of points between iterations. Doing even a few too many iterations could be computationally exhaustive, whereas doing too few may result in a poor approximation of the attractor. Luckily, there is another algorithm which is much more stable with respect to the number of points.

Definition 2.16 (The Random Algorithm / The Chaos Game)**.** *Let* $(X,\, d)$ *be a complete metric space on which the functions in an IFS,* $W = \{\hat{w}_i\}_{i=1}^N$*, are defined. Then choosing an*

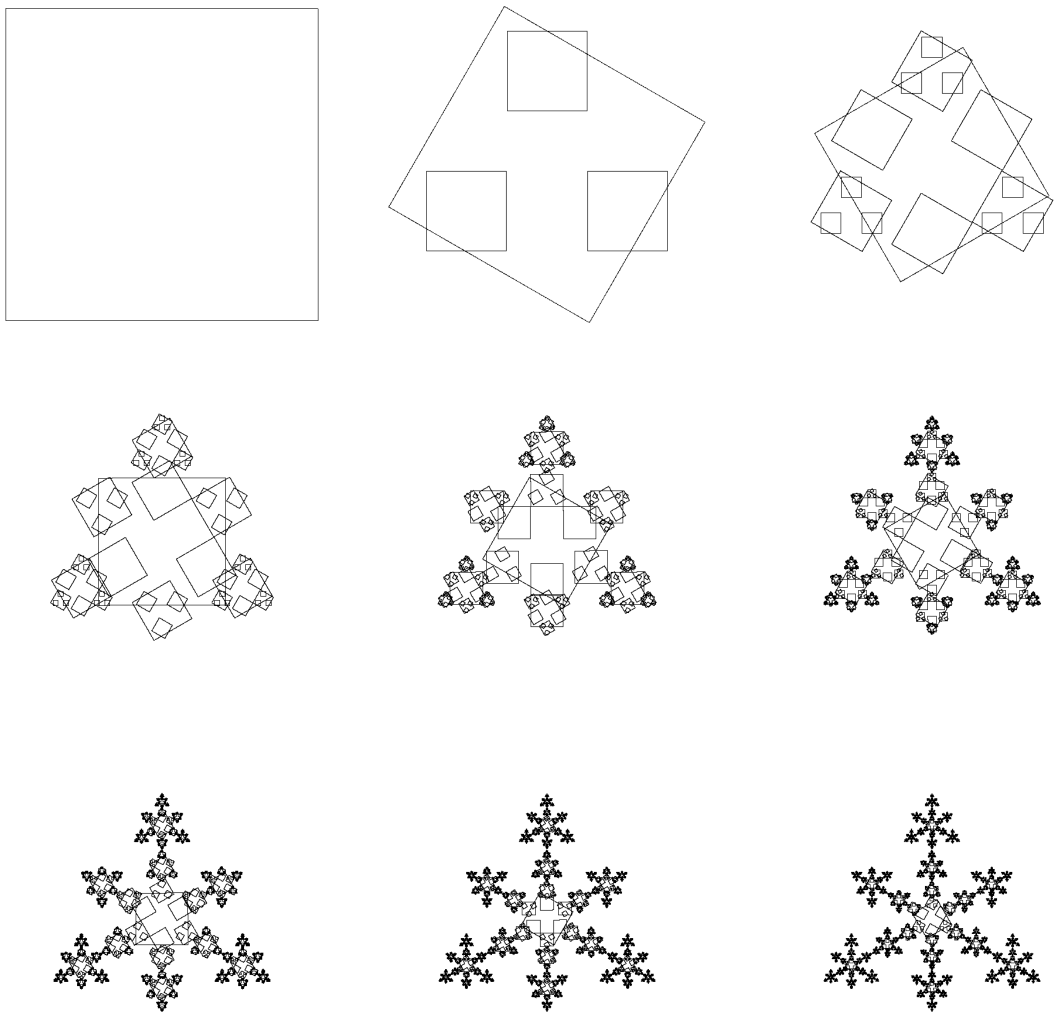


Figure 2.5: The first nine iterations of the deterministic algorithm to generate the attractor of the IFS in Ex. 2.4 where the initial set is the unit square.

initial point $x_0 \in X$, we find the next point by computing

$$x_{n+1} = \hat{w}_{\sigma_n}(x_n),$$

where $\sigma_n \in \{1, 2, \ldots, N\}$ is a random index. The set $\{x_k\}$ is then our approximation to the attractor.

This random algorithm, commonly referred to as the chaos game, is much more frequently used to generate pictures of attractors from IFSs. It allows us to generate the points just as easily as in the deterministic algorithm, but we can choose exactly how many points we would like there to be in our end result. Additionally, we can make this algorithm more efficient than the deterministic algorithm by assigning each function in the given IFS a probability proportional to the size of the contraction factor for a given function. That is, those functions with a larger contraction factor have a higher probability of being selected. The combination of the IFS and probabilities is often called an IFS with Probabilities (IFSP). However, the method being used in this algorithm to find the attractor is not the same as the method used in Banach's fixed point theorem, so we must find a way to show that the sequence of points that are generated do indeed converge to the attractor.

Theorem 2.5. *Let (X, d_X) be a complete metric space on which the IFS $W = \{\hat{w}_i\}_{i=1}^{N}$ is defined with attractor A. Then the sequence $\{x_n\}_{n=0}^{\infty}$ obtained from the random algorithm with indices $\{\sigma_n\}_{n=1}^{\infty}$ contains points arbitrarily close to A.*

Proof. The point-set distance between the first two points in the sequence and the attractor is given by

$$d_{X,\,point}(x_0,\, A) = \inf_{y \in A} d_X(x_0,\, y),$$

and

$$d_{X,\,point}(x_1,\, A) = \inf_{y \in A} d_X(\hat{w}_{\sigma_1}(x_0),\, y).$$

Let A_{σ_1} represent $\hat{w}_{\sigma_1}(A)$, which is clearly a subset of A. Then we see,

$$\begin{aligned}
d_{X,\,point}(x_1,\,A) &\leq \inf_{y\,\in\,A_{\sigma_1}} d_X(\hat{w}_{\sigma_1}(x_0),\,y) \\
&= \inf_{z\,\in\,A} d_X(\hat{w}_{\sigma_1}(x_0),\,\hat{w}_{\sigma_1}(z)) \\
&\leq c \inf_{z\,\in\,A} d_X(x_0,\,z) \\
&= c\, d_{X,\,point}(x_0,\,A).
\end{aligned}$$

where $c = \max\{c_i\}$, and c_i is the contraction factor for $\hat{w}_i$. Then for the k^{th} point in the sequence we get a distance to the attractor of

$$\begin{aligned}
d_{X,\,point}(x_k,\,A) &\leq c\, d_{X,\,point}(x_{k-1},\,A) \\
&\leq c^2\, d_{X,\,point}(x_{k-2},\,A) \\
&\;\;\vdots \\
&\leq c^k\, d_{X,\,point}(x_0,\,A).
\end{aligned}$$

Since $c \in [0,\,1)$, $d_{X,\,point}(x_n,\,A) \to 0$ as $n \to \infty$. Hence the points in the sequence $\{x_n\}$ become arbitrarily close to A. Additionally, we see that for any $\epsilon > 0$, $\exists\, N$ such that if $n > N$, then $d_{X,\,point}(x_n,\,A) < \epsilon$. $\square$

Typically, the first few points in the random algorithm are not in the attractor set. What this theorem allows us to do, though, is exclude from our approximation any points until they are sufficiently close that they are within some specified tolerance (for example, the width of a pixel on a computer screen). Then, all points in the sequence following the last excluded one must be closer to the attractor than it.

Example 2.5. *In this example we approximate a fractal known as the Barnsley fern using the random algorithm with a random first point. We generate the first hundred points, discard*

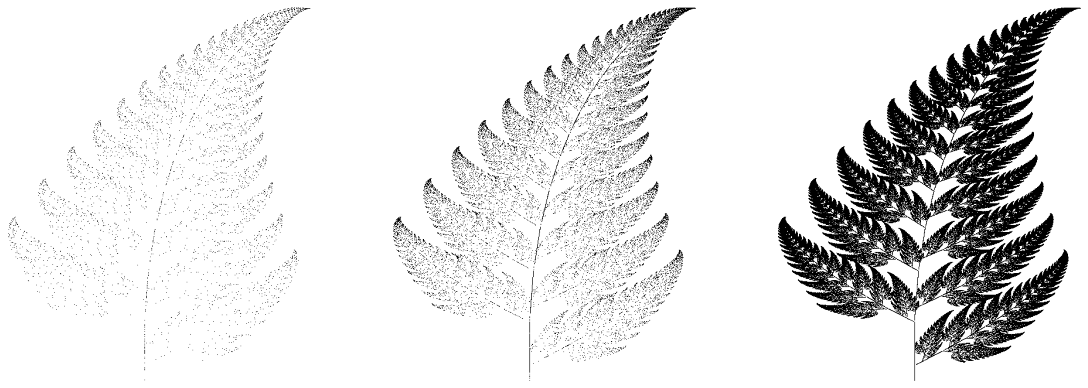


Figure 2.6: The Barnsley fern attractor generated using the random algorithm with 5,000, 30,000, and 1,000,000 points going from left to right respectively.

all but the last, and calculate the desired number of points to obtain the approximation. Using 5,000, 30,000, and 1,000,000 points to approximate the attractor of the IFS below results in the images found in Fig. 2.6.

$$W = \begin{cases} \hat{w}_1(x,\, y) & = \begin{bmatrix} 0 & 0 \\ 0 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad p = 0.01 \\ \hat{w}_2(x,\, y) & = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix} \qquad p = 0.85 \\ \hat{w}_3(x,\, y) & = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6 \end{bmatrix} \qquad p = 0.07 \\ \hat{w}_4(x,\, y) & = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0.44 \end{bmatrix} \qquad p = 0.07 \end{cases}$$

2.4 Fractal Dimension

How can we measure these fractals? Is there some measurement that we can use to compare the attractor in Fig. 2.5 to the one in Fig. 2.6? We cannot compare the area they seem to span because in actuality, both of these attractors span zero area. They only seem to have a non-zero area because we are approximating the attractors and plotting them on a computer with finite resolution. Indeed these fractal sets do not cover an area, yet they cannot be represented with a single dimensional length either; they exist in a non-integer, fractional dimension. An infinite set possessing a non-integer dimension such as this is one definition of a fractal. This section will define what the fractal, or Hausdorff dimension is, and how one can approximate it with only an approximation of an attractor.

We can begin to define this property by noting that it is related to how an object scales. That is, when $[0,\, a]$ is scaled to $[0,\, ba]$, it has b^1 times the length; when $[0,\, a]^2$ is scaled to $[0,\, ba]^2$, it has b^2 times the area. We will define the fractal dimension to be an invariant quantity when scaling a set, such as the one and two in the exponent of these examples. Let $(X,\, d)$ be a complete metric space with $A \in \mathcal{H}(X)$. Let $\epsilon > 0$ and $B(x,\, \epsilon)$ denote a closed ball of radius ϵ centered at the point $x \in X$. Finally, define $\mathcal{N}(A,\, \epsilon)$ to be the smallest positive integer M such that

$$A \subset \bigcup_{n=1}^{M} B(x_n,\, \epsilon),$$

where $\{x_n\}_{n=1}^{M} \subset X$. Note that we know $\mathcal{N}(A,\, \epsilon)$ exists as A is compact, and hence has a finite subcover. We can then define the fractal dimension of A to be D, if for some positive constant C we have

$$\mathcal{N}(A,\, \epsilon) \approx C\epsilon^{-D}.$$

Here "$\approx$" signifies that if $f(\epsilon) \approx g(\epsilon)$, then $\lim\limits_{\epsilon \to 0}(\ln{(f(\epsilon))}/\ln{(g(\epsilon))}) = 1$. Thus, isolating for

D gives us

$$D \approx \frac{\ln(\mathcal{N}(A,\,\epsilon)) - \ln(C)}{\ln(1/\epsilon)}$$

Taking the limit as we decrease the radius of the balls to zero gives us a more formal definition, found in [10].

Definition 2.17 (Fractal Dimension)**.** *Let $(X,\,d)$ be a complete metric space and $A \in \mathcal{H}(X)$. For any $\epsilon > 0$, let $\mathcal{N}(A,\,\epsilon)$ be the smallest number of closed balls of radius ϵ needed to form a cover for A. Then if*

$$D = \lim_{\epsilon \to 0}\left\{\frac{\ln(\mathcal{N}(A,\,\epsilon))}{\ln(1/\epsilon)}\right\}$$

exists, D is the fractal dimension of A.

Since we are taking the limit as the radius of the balls reduces to zero, it does not actually matter whether we use a closed ball, or some other closed set whose size is determined by ϵ. This makes this property of fractals much more easy to calculate computationally.

Example 2.6. *In this example we calculate the fractal dimension of the Sierpiski triangle by splitting each region into boxes at a consistent rate, and counting how many boxes contain a piece of the fractal. As we can see from the images in Fig. 2.7, each time we decrease the side-length of the box by a factor of two, we get three times as many boxes containing a piece of the fractal. Thus, if $\epsilon = (1/2)^k$, then $\mathcal{N}(A,\,\epsilon) = 4 \cdot 3^k$. This allows us to calculate the*

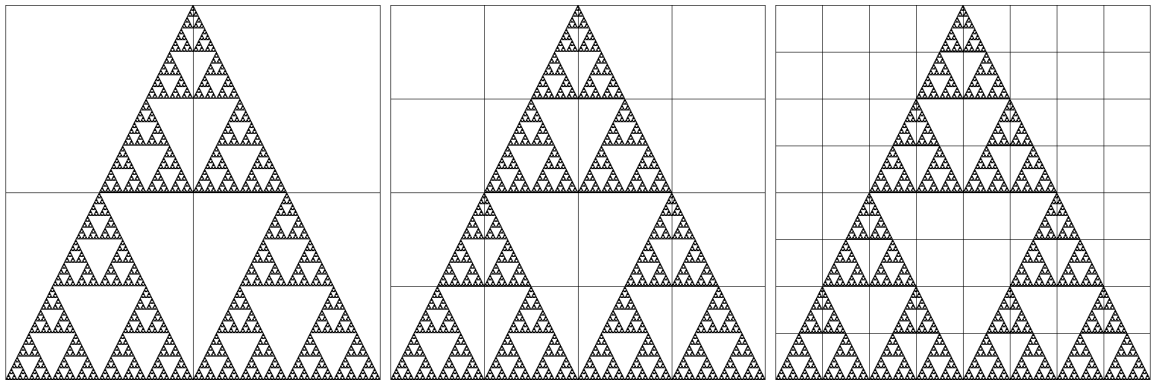


Figure 2.7: Images demonstrating how one can calculate the fractal dimension of the Sierpinski triangle through breaking each region into boxes.

fractal dimension of the Sierpinski triangle as follows:

$$\begin{aligned} D &= \lim_{k\to\infty} \frac{\ln(4\cdot 3^k)}{\ln(\frac{1}{(1/2)^k})} \\ &= \lim_{k\to\infty} \frac{\ln(4^{\frac{1}{k}}\cdot 3)}{\ln(2)} \\ &= \frac{\ln(3)}{\ln(2)} \\ &\approx 1.5849. \end{aligned}$$

Unfortunately, it is typically much more difficult to accurately calculate the fractal dimension, especially as we only have an approximation of the true fractal set. However, we can estimate the fractal dimension by using each pixel as a box. Therefore, for any given fractal we can estimate the fractal dimension by

$$D = \frac{\ln(\#\text{ of pixels corresponding to the fractal})}{\ln(\text{picture width})}.$$

The Sierpinski triangle shown in Fig. 2.7, for example, is a 640×640 image and there are 36446 pixels corresponding to the attractor. Thus, our estimate of the fractal dimension is

Resolution	Estimated Fractal Dimension
160×160	1.6361.585
320×320	1.630
640×640	1.626
1280×1280	1.621

Table 2.1: Estimated fractal dimension for the Sierpinski triangle at various resolutions.

1.626, which is not far off considering the ease with which it was calculated. Additionally, we can see that this approximation tends towards the true value of the fractal dimension as the resolution of the image increases. This is demonstrated in Table 2.1, showing approximations of the fractal dimension of the Sierpinski triangle at various resolutions.

Chapter 3

Creating Databases of Randomly Initialized Fractals

One of the goals of this thesis is to obtain a mapping from an image of an attractor to its parameters through the use of a neural network. This will require vast amounts of data. This chapter will focus on creating that data and analyzing some of its properties.

3.1 Iterated Function Systems with Random Parameters

In this section we will cover how to generate fractals with random parameters. It is not as simple as randomly initializing an IFS, as in order to have an attractor, an IFS must satisfy Banach's fixed point theorem. Thus, the IFS must consist entirely of contractive functions. Therefore, if we randomly initialize parameters of an IFS, we must find a way to check whether each of its functions is contractive before we attempt to generate the attractor. We limit ourselves to fractals represented in two dimensions created from affine transformations.

Thus, a given function in these IFSs can be denoted as

$$\hat{w}(x) = Ax + b,$$

where $A \in \mathbb{R}^{2\times 2}$, and w, x, b, $\in \mathbb{R}^2$. In order for this transformation to be contractive, we require

$$\begin{aligned}
||\hat{w}(x) - \hat{w}(y)||_2 &\leq c||x - y||_2,\ c \in [0,\ 1) \\
||Ax + b - Ay + b||_2 &\leq c||x - y||_2 \\
||A||_2 &\leq c,
\end{aligned}$$

where $||A||_2$ is the induced two-norm of the matrix. Specifically, this norm is defined as

$$\begin{aligned}
||A||_2 &= \sup_{x,\, ||x||_2 \neq 0} \frac{||Ax||_2}{||x||_2} \\
||A||_2^2 &= \sup_{x,\, ||x||_2 \neq 0} \frac{x^\intercal A^\intercal A x}{x^\intercal x} \\
&= \lambda_{max}(A^\intercal A),
\end{aligned}$$

where $\lambda_{max}(A^\intercal A)$ is the largest eigenvalue of $A^\intercal A$. Since we know that $A^\intercal A$ will be a positive semi-definite, and symmetric matrix, all of its eigenvalues will be nonnegative. Recall that the characteristic polynomial of a matrix is the lowest order polynomial whose roots are precisely at the eigenvalues of its corresponding matrix. Let the characteristic polynomial of $A^\intercal A$ be denoted by

$$f(t) = t^2 - \mathrm{Tr}(A^\intercal A)t + \det(A^\intercal A).$$

Applying the restriction $f(1) > 0$ then forces both eigenvalues to be either larger than one, or smaller than it, as this quadratic always has a positive second derivative. In addition, we impose the condition that this should be a contractive function, hence $||Ax||_2 < ||x||_2$. Pairing this last requirement with the standard basis for $\mathbb{R}^2$, we get the following set of conditions to check for contractivity:

$$\begin{cases} a_{1,1}^2 + a_{2,1}^2 & < 1 \\ a_{1,2}^2 + a_{2,2}^2 & < 1 \\ Tr(AA^\intercal) - det(A)^2 & < 1 \end{cases}$$

where $a_{i,j}$ are parameters in the matrix A. We can see that this forces both eigenvalues to be less than one as whether or not the eigenvalues are less than or greater than one is determined by the minimum of the quadratic. This minimum is given by

$$\begin{aligned} f'(t) &= 2t - \mathrm{Tr}(A^\intercal A) = 0 \\ t &= \frac{\mathrm{Tr}(A^\intercal A)}{2} \\ &= \frac{a_{1,1}^2 + a_{1,2}^2 + a_{2,1}^2 + a_{2,2}^2}{2} \\ &< 1 \end{aligned}$$

Example 3.1. *In this example we show some fractals generated using random parameters. The image on the left of Fig. 3.1 is the attractor of the function system:*

$$W = \begin{cases} \hat{w}_1(x,\, y) & = \begin{bmatrix} 0.325482 & 0.651813 \\ -0.843093 & 0.053819 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.128834 \\ 0.109288 \end{bmatrix} \\ \hat{w}_2(x,\, y) & = \begin{bmatrix} 0.537771 & -0.189520 \\ -0.604284 & -0.303044 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.402950 \\ -0.195668 \end{bmatrix} \end{cases}$$

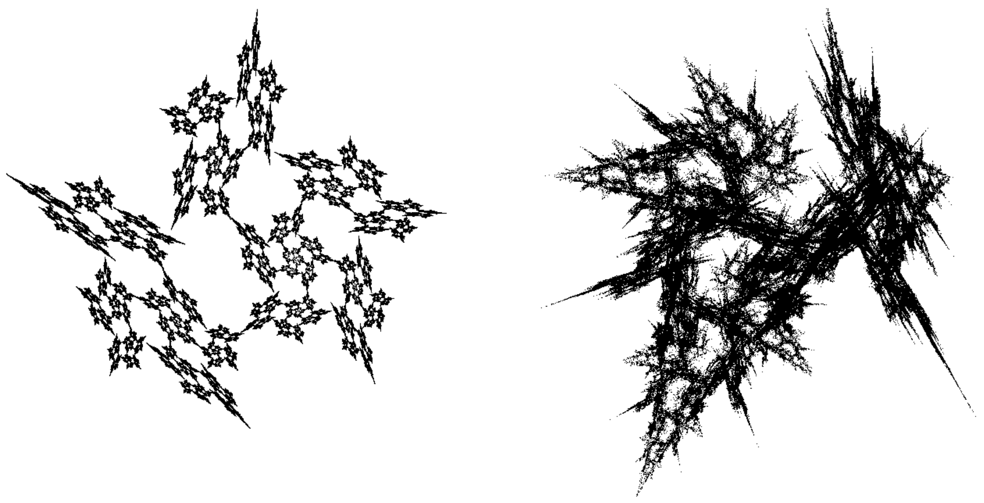


Figure 3.1: Examples of attractor sets pertaining to IFSs with randomly initialized parameters. These fractals were generated using 1,000,000 points with the random algorithm.

The image on the right of Fig. 3.1 has the IFS

$$\begin{cases} \hat{w}_1(x,\, y) &= \begin{bmatrix} -0.4193 & -0.0270 \\ -0.4613 & -0.2053 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.2791 \\ 0.0481 \end{bmatrix} \\ \hat{w}_2(x,\, y) &= \begin{bmatrix} -0.3939 & 0.4496 \\ 0.4406 & 0.4919 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.2792 \\ -0.2756 \end{bmatrix} \\ \hat{w}_3(x,\, y) &= \begin{bmatrix} 0.1568 & -0.3538 \\ -0.7952 & 0.3318 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.4873 \\ 0.2117 \end{bmatrix} \\ \hat{w}_4(x,\, y) &= \begin{bmatrix} 0.3812 & 0.5316 \\ -0.5675 & 0.2246 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.1638 \\ 0.2334 \end{bmatrix} \end{cases}$$

Of all the images of fractals shown so far, none of them have included axes displaying their size, despite all the points being values in $\mathbb{R}^2$. This was done not only to make the pictures look cleaner, but because the edges of most of these images has been fixed as the square of side length two, centered at the origin. However, many of these fractals were not originally able to fit in this region. We can instead randomly initialize a fractal, and later scale its size by noting how the size of an attractor is affected by an IFSs parameters. Define the sequence

$$\begin{aligned}
x_{n+1} &= \hat{w}(x_n) \\
&= Ax_n + b \\
&= A(Ax_{n-1} + b) + b \\
&\vdots \\
&= A^n x_0 + \sum_{k=0}^{n-1} A^k b.
\end{aligned}$$

In the limit as $n \to \infty$ we see that the first term vanishes since $||A||_2 < 1$, and the sum approaches a constant. Thus $||x_n||_2$ is proportional to $||b||_2$. We can thus change the size of a fractal by scaling all parameters in b of each function in an IFS. We will call these components of each function the **additive parameters**.

Example 3.2. *In this example we randomly initialize a four function IFS, and scale its attractor to fit in the square of side length two centered at the origin. We randomly sampled the* 24 *parameters from a uniform distribution between the values* -1 *and* 1*. After ensuring contractivity we get the IFS*

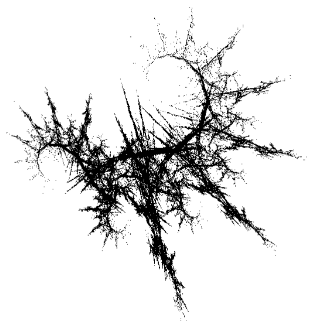


Figure 3.2: The attractor corresponding to the randomly generated IFS in Ex. 3.2

$$W = \begin{cases} \hat{w}_1(x,\, y) &= \begin{bmatrix} -0.280024 & -0.416243 \\ 0.552090 & -0.103137 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.897841 \\ 0.414642 \end{bmatrix} p = 0.25 \\ \hat{w}_2(x,\, y) &= \begin{bmatrix} -0.243396 & 0.371675 \\ -0.013273 & 0.078729 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.077269 \\ 0.673438 \end{bmatrix} p = 0.25 \\ \hat{w}_3(x,\, y) &= \begin{bmatrix} -0.210925 & 0.150266 \\ 0.827353 & 0.081144 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.024475 \\ -0.452672 \end{bmatrix} p = 0.25 \\ \hat{w}_4(x,\, y) &= \begin{bmatrix} 0.664901 & -0.423435 \\ 0.444192 & 0.767060 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.991207 \\ 0.362855 \end{bmatrix} p = 0.25 \end{cases}$$

where p is the probability of selection for each function. After generating the first 1,000,000 points using the random algorithm, the attractor set was not contained in the desired region. Specifically, it reached in magnitude a value of 3.16 *in the x direction, and* 3.66 *in the y direction. To correct this, all additive parameters were divided by the larger of these two values,* 3.66*, resulting in the new absolute maxima of* 0.85 *and* 0.97 *in the x and y directions respectively. Note that it does not reach a magnitude of exactly one as both our initial set for the attractor and the one generated after scaling the additive parameters are only approximations. The attractor of this IFS is shown in Fig. 3.2.*

One important advantage of putting each fractal in a fixed universal viewing window such as this, is that the value of a pixel does not change between images. This will be pertinent when examining databases we create with these fractals, as well as when we use them to train a neural network to predict their parameters.

3.2 Properties of Randomly Generated Fractal Sets

Using the methods shown in the previous section, we generated fractal databases of attractors made from IFSs with each of two, four, six, and eight functions, by doing the following:

1. Randomly initialize an IFS with parameters sampled from a uniform distribution spanning $[-1, 1]$, and equally distributed probabilities.
2. Check all functions of the IFS for contractivity, and regenerate all parameters of the functions that are not contractive.
3. Generate 1,000,100 points using the random algorithm, and discard the first hundred.
4. Check the generated points for values that lie outside $[-1, 1]^2$. If no points are found, store the fractal in the database. If one or more points is found, continue to the next step.
5. Divide all additive parameters in the IFS by the largest single coordinate value, and return to step three.

The resolution of the images was chosen to be 640×640. While a higher resolution could provide more precise details of the attractor, images of this size are much larger than those of typical neural network applications. As we will see in chapter five, this is because neural networks can take up large amounts of memory, and the higher the resolution, the more memory will be required. Two questions that may arise, though, are how do we know

Functions in the IFS	2	4	6	8
Avg. Absolute Difference	58.0	486.22	966.36	1215.82
Avg. Relative Difference (%)	0.19	1.12	1.62	1.66

Table 3.1: Comparison of the number of pixels corresponding to an attractor when it is generated with 1,000,000 points of the random algorithm, to when it is generated with $1,100,000$ points.

1,000,000 points is sufficient to represent the attractor with this resolution? And, how do we know 100 points is sufficient to remove points that are not within the width of a pixel? These are especially pertinent when considering that the probability associated with each function was evenly distributed, as we do not know the optimal values. The answer to the first question is the following: Examining these databases will show that the number of pixels used to represent each fractal is almost always less than 200,000, regardless of the number of functions in the IFS. Thus, many of the pixels represent a multitude of points, and generating more would not drastically change the resultant image. Empirical evidence of this is shown in Table 3.1, where we generated 1,000 randomly initialized fractals with 1,000,000 points, regenerated them with 1,100,000 points, and compare the number of pixels corresponding to the attractor. To answer the second question, consider a relationship we derived in the previous chapter:

$$d_{X,point}(x^k,\, A) \leq c^k d_{X,point}(x_0,\, A),$$

where c is the largest contraction factor of a function in the IFS. To ensure that the hundredth point is within the width of a pixel no matter the initial point and location of the attractor, we require

$$c^{100} < \frac{1}{640} \rightarrow c < 0.937.$$

Though, when considering that the other functions are likely to be more contractive, discarding the first hundred points is likely to be enough for the vast majority of randomly generated fractals. Additionally, we will see in Chapter 5 that a few pixels out of place will likely not affect the output of a neural network significantly.

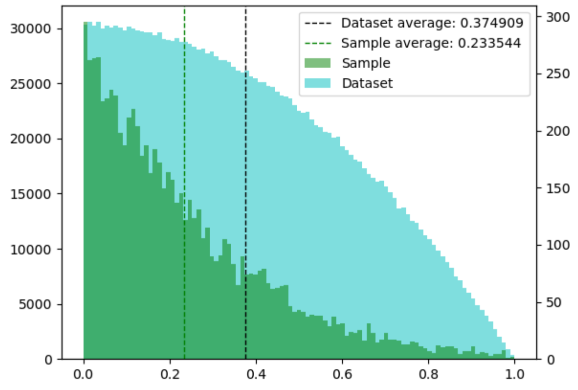
Each of the generated databases contained 250,000 fractals. For each one, not only was the image of the attractor stored, but the following variables as well:

- All fractal parameter values
- The number of pixels taken up by the fractal
- An estimate of the fractal dimension
- The centroid of the fractal pixels
- The standard deviation of the fractal pixels from the centroid in both the horizontal and vertical directions

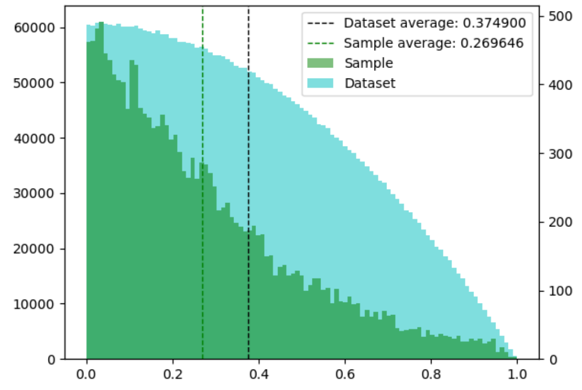
A detail that will become important when training a neural network with these randomly generated IFSs is the way the parameters are stored. Since the attractor of an IFS will be the same no matter the order of the functions, we have to be able to account for this when training a neural network to predict parameters of an IFS for a given image. So, the functions were ordered based on the value of the parameters in the matrix of the affine transformation. Labelling these parameters as

$$w(x,\, y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p \\ q \end{bmatrix},$$

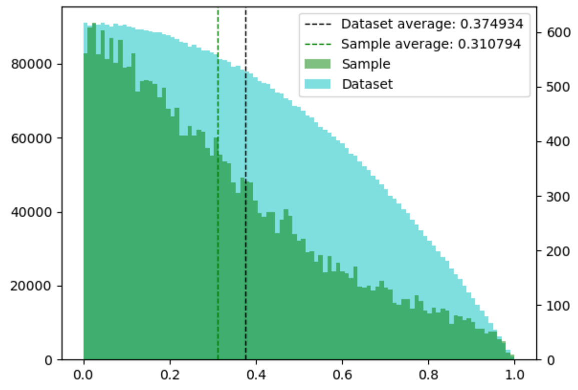
then the functions were ordered in increasing order of the a parameters. If two functions were to possess the same value, an unlikely case for randomly generated double precision numbers, then they were ordered in increasing order of their b parameter. Saving all these variables allows us to analyze properties of fractals in a more general setting, and view trends as the number of functions in a given IFS is changed. Fig. 3.3 shows comparisons of a subset of a given dataset relative to the entire respective dataset. The values being compared are the absolute values of the **multiplicative parameters**; those parameters in the matrix of the affine transformation, where the subset is the top ∼1,000 fractals with the fewest pixels pertaining to the attractor. What we can deduce from this set of graphs, is that for IFSs comprised of fewer functions, the magnitude of the multiplicative parameters has a greater influence on the number of pixels used to represent the fractal. The same trend is seen in Fig. 3.4, which compares the same property, but the subset is constructed from the top ∼1,000 fractals that have the most attractor pixels. It makes sense that the absolute size of these multiplicative parameters will partially determine the number of pixels required to represent the attractor as they are related to each functions contraction factor. Fractals with larger parameters will be less contractive, and hence span more pixels. To understand why this effect dies out with more functions, we must first look at some other properties.



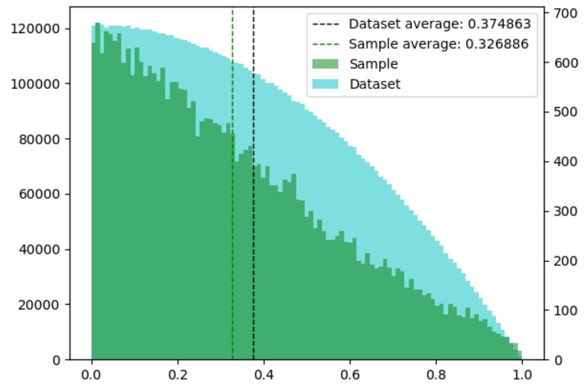
(a)



(b)

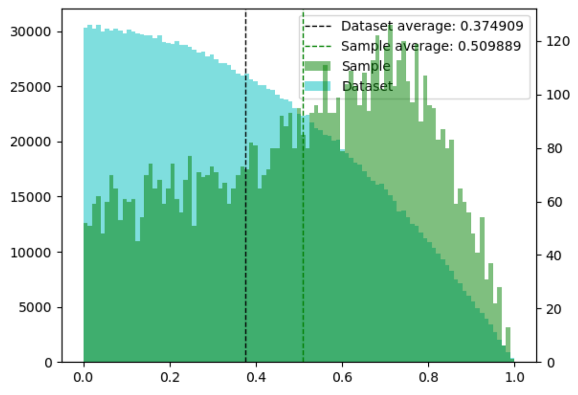


(c)

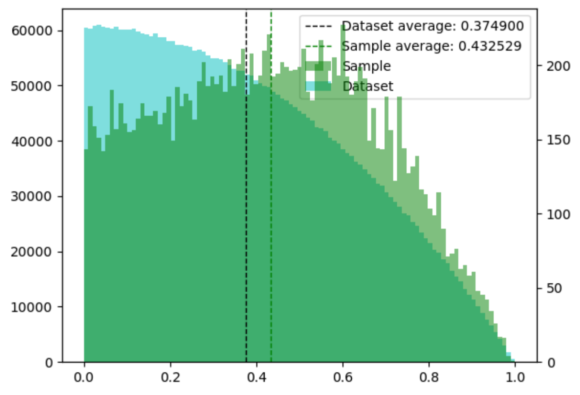


(d)

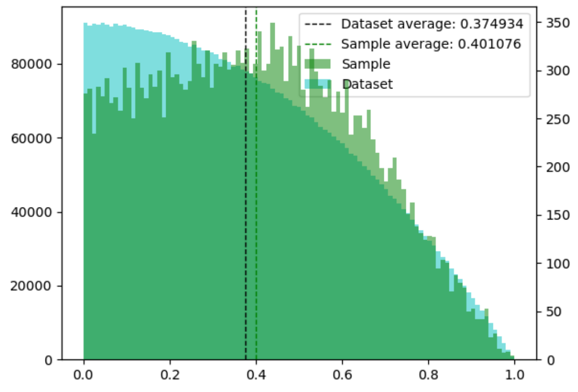
Figure 3.3: Comparing distributions of the absolute value of the multiplicative parameters. The green represents the top ∼1,000 attractors for being composed of the fewest amount of pixels. The blue is the distribution of the entire dataset from which the green subset was taken. (a), (b), (c), and (d) shows the distributions for the databases of fractals with two, four, six, and eight functions in their IFSs respectively.



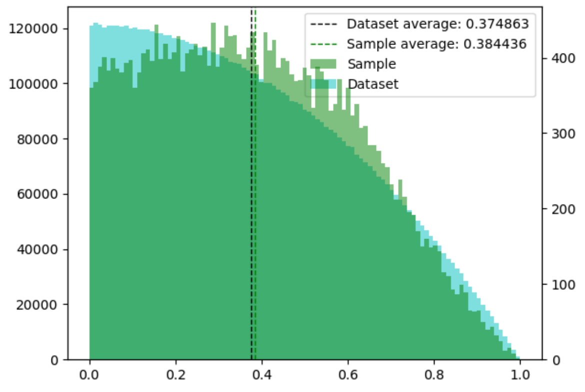
(a)



(b)



(c)

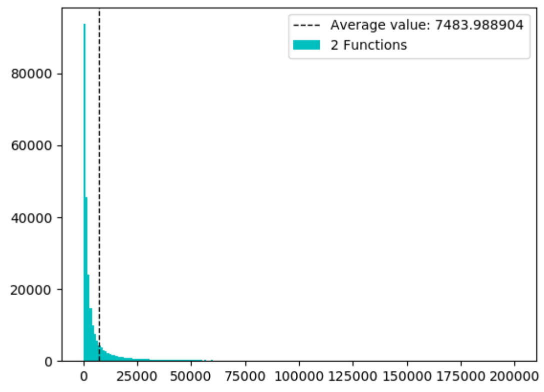


(d)

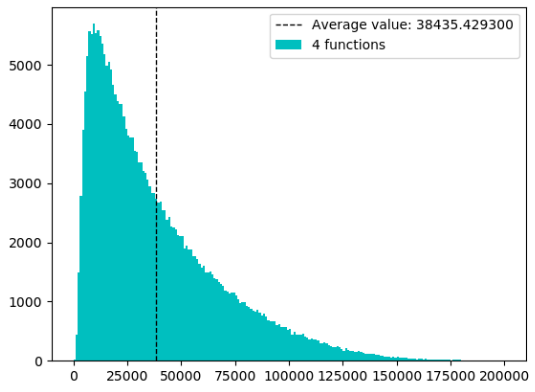
Figure 3.4: Comparing distributions of the absolute value of the multiplicative parameters. The green represents the top ∼1,000 attractors for being composed of the most amount of pixels. The blue is the distribution of the entire dataset from which the green subset was taken. (a), (b), (c), and (d) shows the distributions for the databases of fractals with two, four, six, and eight functions in their IFSs respectively.

The next properties we will examine are the effect the number of functions in an IFS has on the number of pixels corresponding to the fractal, and the standard deviation with respect to the centroid of those pixels in the horizontal direction. These are displayed in Fig. 3.5 and Fig. 3.6 respectively. Note that the standard deviation with respect to the centroid in the vertical direction produces similar results. As we can see from these images, the more functions that make up a given IFS, the more pixels it will require to represent its attractor, on average. We can also see from Fig. 3.6, that the standard deviation decreases, on average, for IFSs comprised of increasing amounts of functions. The overarching theme, then, is that as we increase the number of functions in an IFS, on average we will go from obtaining a fractal with fewer pixels that is more spread out, to one with more pixels that is more dense around its centroid. This makes sense when considering how the random algorithm works to generate each fractal. Each function in the IFS has a globally attractive fixed point. So, when the fractal is generated in the random algorithm, whichever function is chosen as the next transformation will pull the input point slightly closer to its own fixed point. As we increase the number of functions in the IFS, there are more possible fixed points that the input point could get pulled towards, leading to a higher concentration of pixels around the average location. This is also the reason why with more functions, the absolute size of the multiplicative parameters plays less of an effect on the number of pixels corresponding to the fractal.

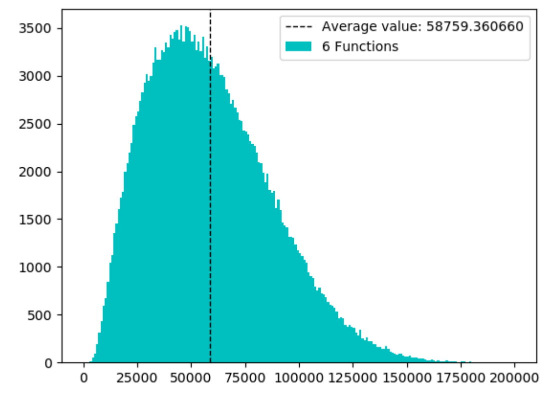
The last properties that we can examine are the centroid location and fractal dimension. Though, we will not include images of these plots for the following reasons: The centroid location was approximately normally distributed for all databases. The fractal dimension is a monotonic transformation of the number of pixels used to represent the attractor. One can attain the plots for the fractal dimension by using a log-scaled x-axis, and scaling the data of the images in Fig. 3.5.



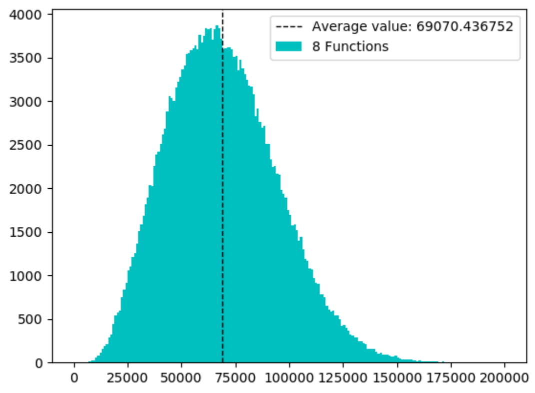
(a)



(b)

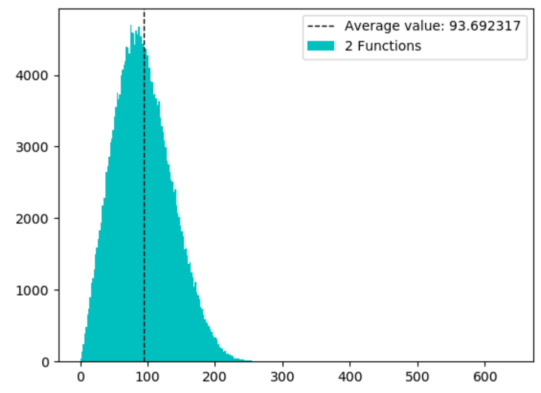


(c)

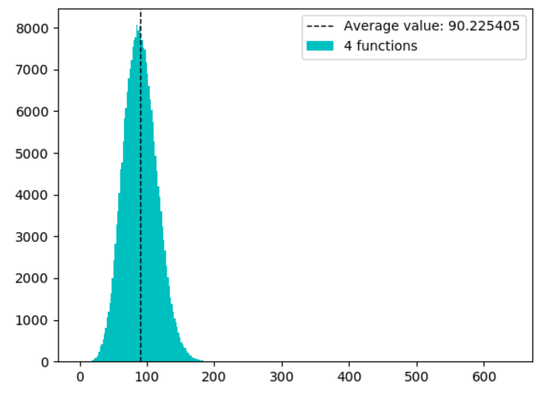


(d)

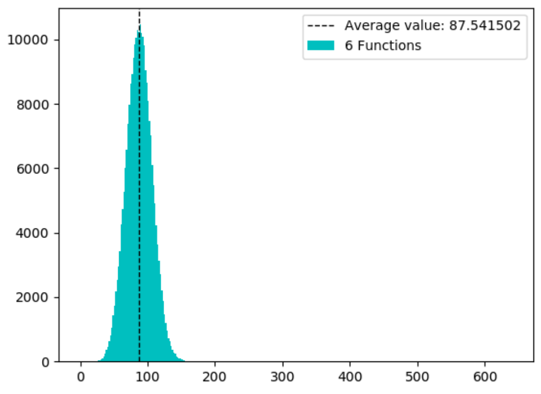
Figure 3.5: This figure shows the distribution of the number of pixels each fractal possesses for multiple datasets. (a), (b), (c), and (d) show the distributions for the databases of fractals with two, four, six, and eight functions in their IFSs respectively.



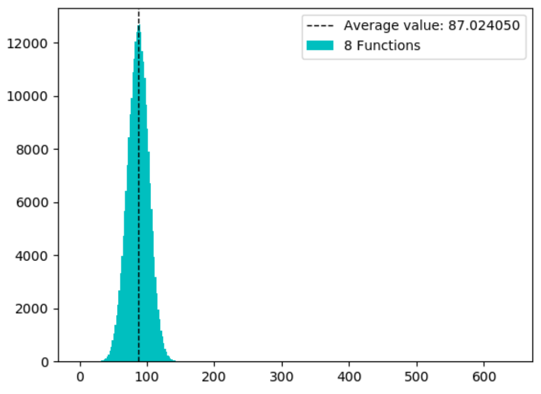
(a)



(b)



(c)



(d)

Figure 3.6: This figure shows the distribution of the standard deviation along the horizontal direction of the pixels each fractal possesses for multiple datasets. (a), (b), (c), and (d) show the distributions for the databases of fractals with two, four, six, and eight functions in their IFSs respectively.

Chapter 4

Fractals Created From Nonaffine Functions

Before continuing to the neural network portion of this thesis, we take a detour to fractals created from IFSs consisting of nonaffine functions. As of yet, we have only discussed attractors created from affine transformations. This is in part due to the fact that there is very little research done on anything else. Approximated attractor sets created in this chapter are purely for curiosity's sake of what they might look like, and what functions may be used to create them; they will not be used beyond this chapter.

Similar to randomly generated affine functions, we must have a set of conditions to guarantee that these nonaffine functions are contractive when randomly generating parameters. Once a function is known to be contractive, it will satisfy Banach's fixed point theorem, as well as the theorems stating that the IFS will be a contraction mapping in $(\mathcal{H}(X),\, d_{\mathcal{H}})$. For each section in this chapter, we will derive these conditions and present some examples of fractals created from the functions that satisfy them. Also note that we arbitrarily chose each IFS in this chapter to be a set of four functions, and all attractors were generated with $1,000,000$ points of the random algorithm, after discarding the first hundred.

4.1 Bounded Derivative Functions

The first group of nonaffine functions we will look at are those for which we can bound the derivative. If we have a function $f(x) = a\,g(x) + c$ such that $|g'(x)| \leq M\,\forall x$, where a, c, $\in \mathbb{R}$ and $M \in \mathbb{R}^+$, then by the mean value theorem $\exists\, p \in (x,\, y)$ such that

$$\begin{aligned}|f(x) - f(y)| &= |x - y|\,|f'(p)| \\ &= |a|\,|x - y|\,|g'(p)| \\ &\leq M\;|a|\,|x - y|\end{aligned}$$

Thus, in order for f to be a contractive function, we require $|a| < 1/M$. We will call these functions represented by g **bounded derivative functions** and denote the set of all of them by G. Extending this to two dimensions, let $x = (x_1,\, x_2)$, $y = (y_1,\, y_2)$, and

$$f(x) = \begin{bmatrix} a_{1,1}g_1(x_1) + a_{1,2}g_2(x_2) \\ a_{2,1}g_3(x_1) + a_{2,2}g_4(x_2) \end{bmatrix} + C$$

where $a_{i,j} \in \mathbb{R}$, $C \in \mathbb{R}^2$, and $g_i \in G$. We will denote the matrix with elements $a_{i,j}$ by A. Checking for contractivity we see,

$$||f(x) - f(y)||_2 = \left\| \begin{bmatrix} a_{1,1}(g_1(x_1) - g_1(y_1)) + a_{1,2}(g_2(x_2) - g_2(y_2)) \\ a_{2,1}(g_3(x_1) - g_3(y_1)) + a_{2,2}(g_4(x_2) - g_4(y_2)) \end{bmatrix} \right\|_2 .$$

Taking the absolute value of each element, and applying the triangle inequality and mean value theorem results in

$$\begin{aligned}
||f(x) - f(y)||_2 &\leq \left\| \begin{bmatrix} |a_{1,1}| \cdot |g_1(x_1) - g_1(y_1)| + |a_{1,2}| \cdot |g_2(x_2) - g_2(y_2)| \\ |a_{2,1}| \cdot |g_3(x_1) - g_3(y_1)| + |a_{2,2}| \cdot |g_4(x_2) - g_4(y_2)| \end{bmatrix} \right\|_2 \\
&\leq \left\| \begin{bmatrix} |a_{1,1}| \cdot |x_1 - y_1||g_1'(p_1)| + |a_{1,2}| \cdot |x_2 - y_2||g_2'(p_2)| \\ |a_{2,1}| \cdot |x_1 - y_1||g_3'(p_3)| + |a_{2,2}| \cdot |x_2 - y_2||g_4'(p_4)| \end{bmatrix} \right\|_2,
\end{aligned}$$

where $p_1,\, p_3 \in (x_1,\, y_1)$, and $p_2,\, p_4 \in (x_2,\, y_2)$. Let M be the largest bound on the derivative of all g_i. Then,

$$\begin{aligned}
||f(x) - f(y)||_2 &\leq \left\| \begin{bmatrix} |a_{1,1}| \cdot |x_1 - y_1|M + |a_{1,2}| \cdot |x_2 - y_2|M \\ |a_{2,1}| \cdot |x_1 - y_1|M + |a_{2,2}| \cdot |x_2 - y_2|M \end{bmatrix} \right\|_2 \\
&\leq M||A'||_2||x - y||_2,
\end{aligned}$$

where A' corresponds to the element-wise absolute value of A. Note that all of the contractivity conditions derived in the previous section were invariant to changes in sign. Hence, to guarantee contractivity, we simply need to check $M||A||_2 < 1$, and can use the same conditions as before. Additionally, notice that this generalization has the same strength as we had with affine transformations, as we could choose $g_i(x) = x\, \forall i$.

We will generate attractors with the g_i functions $\sin(b_i x)$, $\cos(b_i x)$, and $\tanh(b_i x)$, where $b_i \in \mathbb{R}$ is different for each g_i. All of these functions have their derivative bounded by $M_i = b_i$, where $|g_i(x)| \leq M_i \forall x$. For simplicity, we will reduce the number of possible permutations of functions to explore by restricting ourselves with $g_1 = g_3$, and $g_2 = g_4$, apart from the b_i values. Selecting all of the parameters from a uniform distribution between -1 and 1, we generated 100 fractals for each permutation of the functions g_i in the transformation, f,

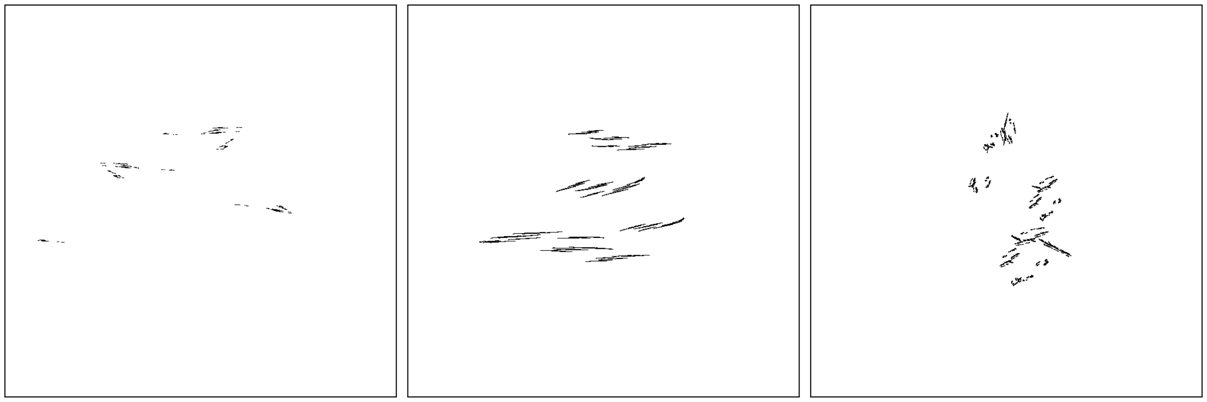


Figure 4.1: Examples of some of the less sparse attractors created from the bounded derivative functions. From left to right shows transformation consisting of the functions cosine and cosine, sine and sine, and hyperbolic tangent and hyperbolic tangent as the g_1 and g_2 functions. The viewing window for each image is $[-3, 3]^2$.

above.

It was realized that the b_i parameters being less than one resulted in much more sparse attractors. That is, the number of pixels that represent the attractor is quite small relative to those created from affine transformations. This effect was slightly inhibited when the g_i functions were chosen to be sine and the hyperbolic tangent. Examples of these attractors are shown in Fig. 4.1.

One cause of this sparsity could be from making the functions more contractive than necessary. Since we randomly selected all parameters to have a magnitude less than one, this may skew the distribution of functions to be more contractive. For example, if $\max_i(|b_i|) < 0.5$ for a particular function, then even if all $a_{i,j}$ were to have a magnitude of 1, a very unlikely case, this would still be a contractive function. We can reduce the extent of this problem by enforcing $M = 1$, and then only checking if $||A|| < 1$, much like we do for affine functions. This can be done by using $g_i^{new}(b_i x) = g_i(b_i x)/b_i$. Note that we could now relax the restriction of $|b_i| \leq 1$, though this will not be done for the following reason: As b_i increases, the range of the functions that we are using decreases, which would increase the sparsity of the attractors. For example, the function $\cos(4x)/4$ can only produce values in $[-0.25, 0.25]$. This restricted range is likely to get further restricted through fixed point iteration. Some examples of attractors created from these functions are given in Fig. 4.2, for more, see the appendix.

We can see that the attractors of IFSs constructed from functions with a derivative bounded by one are much less sparse than before. Additionally, they often display curvature that does not seem to appear in fractals created from affine functions, yet they still maintain the feature of possessing shrunken distorted copied of themselves.

As this new transformation retains all the parameters of an affine transformation, we can make bounded derivative function extensions of common affine fractals. This will by done by maintaining all the parameters in A and C as defined above, randomly selecting $b_i \in [-1, 1]$, and forcing $M = 1$ in the same manner as we did before. Results of this are shown in Fig. 4.3. We can see that these fractals maintain many of their original features. This may be attributed to the fact that for small arguments, both sine, and the hyperbolic tangent are approximated by x. Similarly for cosine, we have that $\cos(b_i(x \mp \pi/2))/b_i \approx \pm x$ for small arguments.

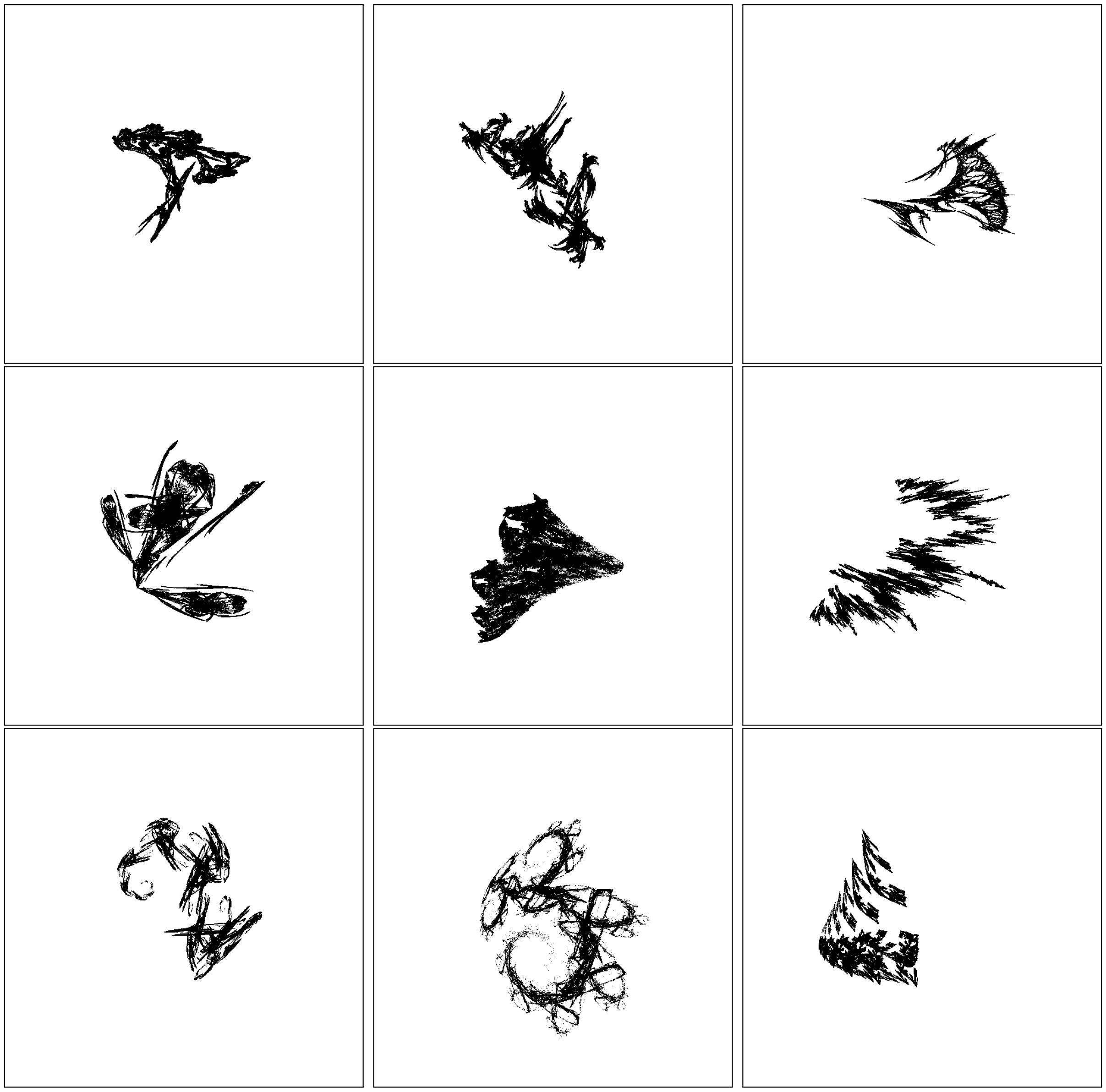


Figure 4.2: Examples of attractors created from bounded derivative functions in which a bound of one was forced by dividing each function by b_i. Each column from left to right has $cos(b_ix)/b_i$, $sin(b_ix)/b_i$, and $tanh(b_ix)/b_i$ as the $g_1 = g_3$ function respectively. Each row from top to bottom has $cos(b_ix)/b_i$, $sin(b_ix)/b_i$, and $tanh(b_ix)/b_i$ as the $g_2 = g_4$ function respectively. The viewing window for each of these is $[-4, 4]^2$.

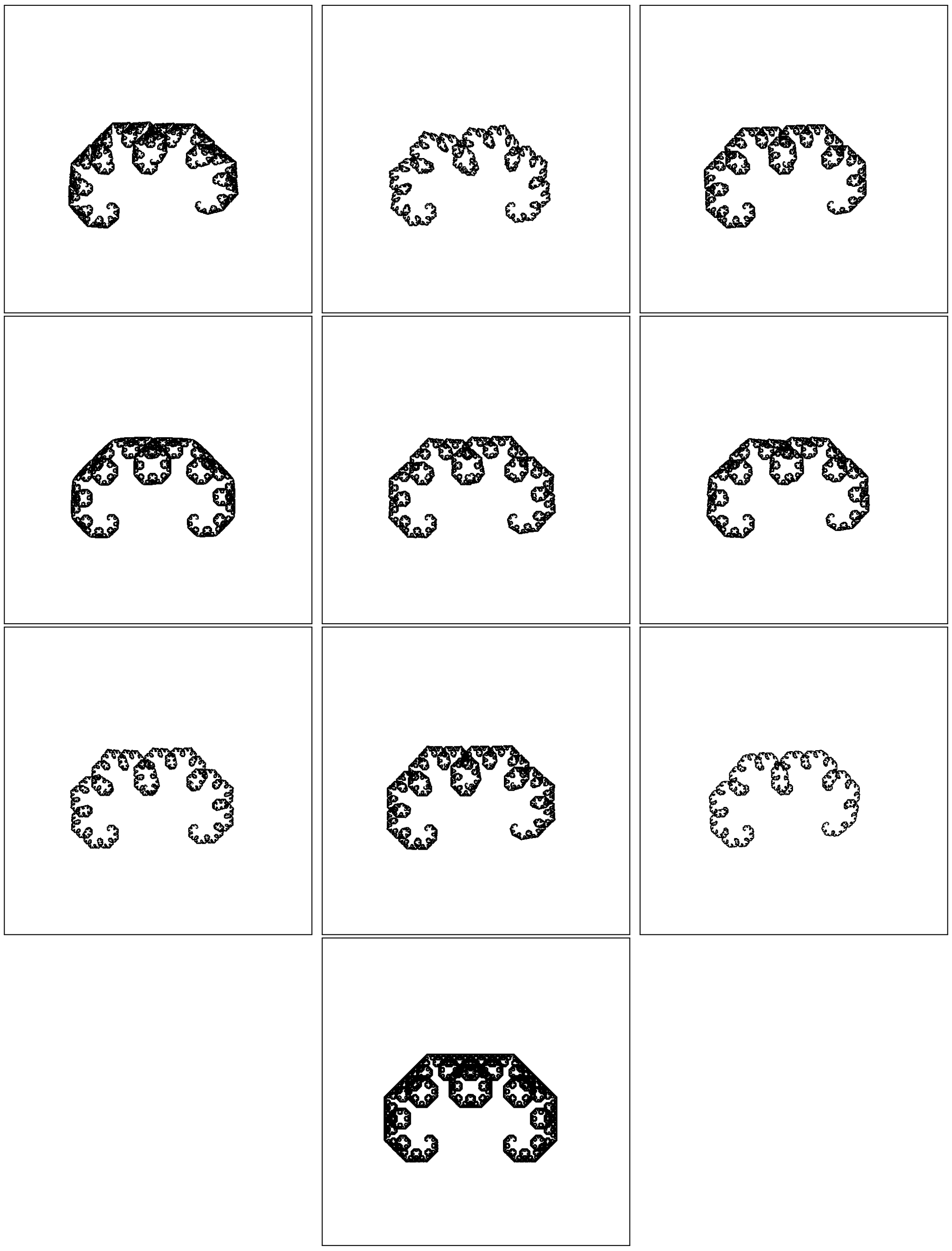


Figure 4.3: Bounded derivative function extensions of the Levy C fractal, shown at the bottom. Each column from left to right has $cos(b_i x)/b_i$, $sin(b_i x)/b_i$, and $tanh(b_i x)/b_i$ as the $g_1 = g_3$ function respectively. Each row from top to bottom has $cos(b_i x)/b_i$, $sin(b_i x)/b_i$, and $tanh(b_i x)/b_i$ as the $g_2 = g_4$ function respectively. The viewing window for each of these is $[-0.75,\, 1.5]^2$.

4.2 Piecewise Functions

In this section we create attractors from piecewise functions. We can effectively have one IFS if a given point in the random algorithm sequence satisfies a specific condition, and a completely different IFS if it does not. Although, we must first show that this piecewise IFS is a contraction mapping in the space $(\mathcal{H}(X),\, d_{\mathcal{H}})$. To do this, we will repeat the proof of Thm. 2.4, where $\hat{w}_1$, one of two functions in an IFS, is now a piecewise function. As before, let $R,\, S \in \mathcal{H}(X)$. Then the distance between these sets after applying the Hutchinson operator is

$$\begin{aligned} d_{\mathcal{H}}(H_W(R),\, H_W(S)) &= d_{\mathcal{H}}(\hat{w}_1(R) \cup \hat{w}_2(R),\, \hat{w}_1(S) \cup \hat{w}_2(S)) \\ &\leq \max\{d_{\mathcal{H}}(\hat{w}_1(R),\, \hat{w}_1(S)),\, d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\}. \end{aligned}$$

Define

$$\hat{w}_1(x) = \begin{cases} \hat{w}_1^1(x) & x \in Q \\ \hat{w}_1^2(x) & otherwise \end{cases},$$

and let $R_1 = R \cap Q$, $R_2 = R \setminus R_1$, $S_1 = S \cap Q$, and $S_2 = S \setminus S_1$. Then we see

$$\begin{aligned} d_{\mathcal{H}}(H_W(R),\, H_W(S)) &\leq \max\{d_{\mathcal{H}}(\hat{w}_1^1(R_1) \cup \hat{w}_1^2(R_2),\, \hat{w}_1^1(S_1) \cup \hat{w}_1^2(S_2)),\, d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\} \\ &\leq \max\{\max\{d_{\mathcal{H}}(\hat{w}_1^1(R_1),\, \hat{w}_1^1(S_1)),\, d_{\mathcal{H}}(\hat{w}_1^2(R_2),\, \hat{w}_1^2(S_2))\}, \\ &\qquad d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\} \\ &\leq \max\{\max\{c_1^1 d_{\mathcal{H}}(R_1,\, S_1),\, c_1^2 d_{\mathcal{H}}(R_2,\, S_2)\},\, d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\}, \end{aligned}$$

where c_1^1 and c_1^2 are the contraction factors of $\hat{w}_1^1$ and $\hat{w}_1^2$ respectively. Since R_1 and R_2, and S_1 and S_2 are completely disjoint,

$$\max\{c_1^1 d_{\mathcal{H}}(R_1,\, S_1),\, c_1^2 d_{\mathcal{H}}(R_2,\, S_2)\} = \max\{c_1^1,\, c_1^2\} d_{\mathcal{H}}(R,\, S),$$

hence we see,

$$\begin{aligned} d_{\mathcal{H}}(H_W(R),\, H_W(S)) &\leq \max\{\max\{c_1,\, c_2\} d_{\mathcal{H}}(R,\, S),\, d_{\mathcal{H}}(\hat{w}_2(R),\, \hat{w}_2(S))\} \\ &\leq \max\{c_1^1,\, c_1^2,\, c_2\} d_{\mathcal{H}}(R,\, S). \end{aligned}$$

Thus, an IFS containing a piecewise function is contractive in the space $(\mathcal{H}(X),\, d_{\mathcal{H}})$ as long as both branches of the piecewise function are contractive. This can also be easily extended to include more than one piecewise function, more than two pieces per piecewise function, and more than two functions in the IFS.

We will limit ourselves to piecewise affine functions throughout this section, as well as only two function pieces per piecewise function. We have not developed a way to scale these fractals like we do with regular affine functions, so the viewing region of attractors throughout this section was chosen to be $[-4, 4]^2$, as it was found to encompass most randomly generated attractors. Since this window does not support the range of all the attractors, though, we generated datasets of 250 fractals, instead of 100 like in the previous section. The IFSs were constructed by randomly selecting their parameters from the uniform distribution spanning $[-1, 1]$, and checking the same affine contractivity conditions as before. Some examples of these piecewise affine IFS attractors are shown in Fig. 4.4. The top row pertains to IFSs with a piecewise condition being whether the x coordinate is less than zero or not, and the bottom row corresponds to IFSs with the piecewise condition being whether the y-value is less than zero or not. For example, the first attractor in the upper row has the function system:

$$W = \begin{cases} \hat{w}_1(x,\, y) = \begin{cases} \begin{bmatrix} 0.1397 & -0.2843 \\ 0.3063 & -0.8336 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.0481 \\ 0.0290 \end{bmatrix} & x < 0 \\ \begin{bmatrix} 0.3599 & 0.4186 \\ -0.0924 & -0.3478 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.0286 \\ -0.1175 \end{bmatrix} & x \geq 0 \end{cases} & p = 0.25 \\ \hat{w}_2(x,\, y) = \begin{cases} \begin{bmatrix} -0.6790 & 0.4759 \\ 0.0393 & 0.6567 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.0620 \\ 0.1412 \end{bmatrix} & x < 0 \\ \begin{bmatrix} 0.1602 & 0.6751 \\ 0.3961 & -0.0220 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.0492 \\ -0.0714 \end{bmatrix} & x \geq 0 \end{cases} & p = 0.25 \\ \hat{w}_3(x,\, y) = \begin{cases} \begin{bmatrix} -0.1891 & -0.5533 \\ -0.6981 & 0.1319 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.0152 \\ -0.3393 \end{bmatrix} & x < 0 \\ \begin{bmatrix} 0.9226 & 0.3412 \\ -0.2114 & 0.7290 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.0598 \\ 0.9776 \end{bmatrix} & x \geq 0 \end{cases} & p = 0.25 \\ \hat{w}_4(x,\, y) = \begin{cases} \begin{bmatrix} -0.8095 & -0.2954 \\ -0.3793 & -0.1487 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.0920 \\ 0.5824 \end{bmatrix} & x < 0 \\ \begin{bmatrix} -0.5152 & -0.6344 \\ 0.6020 & -0.2905 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.9057 \\ 0.4934 \end{bmatrix} & x \geq 0 \end{cases} & p = 0.25 \end{cases}$$

The particular function chosen for this IFS depends on the $x = 0$ line; we will call this the **piecewise boundary**. Similarly, we will refer to the piecewise components of the IFS defined for one particular side of the piecewise boundary as a **branch** of the IFS. With this terminology, there are some interesting things about the attractors given in Fig. 4.4. To begin with, there is no definitive lines along the piecewise boundaries, showing where the

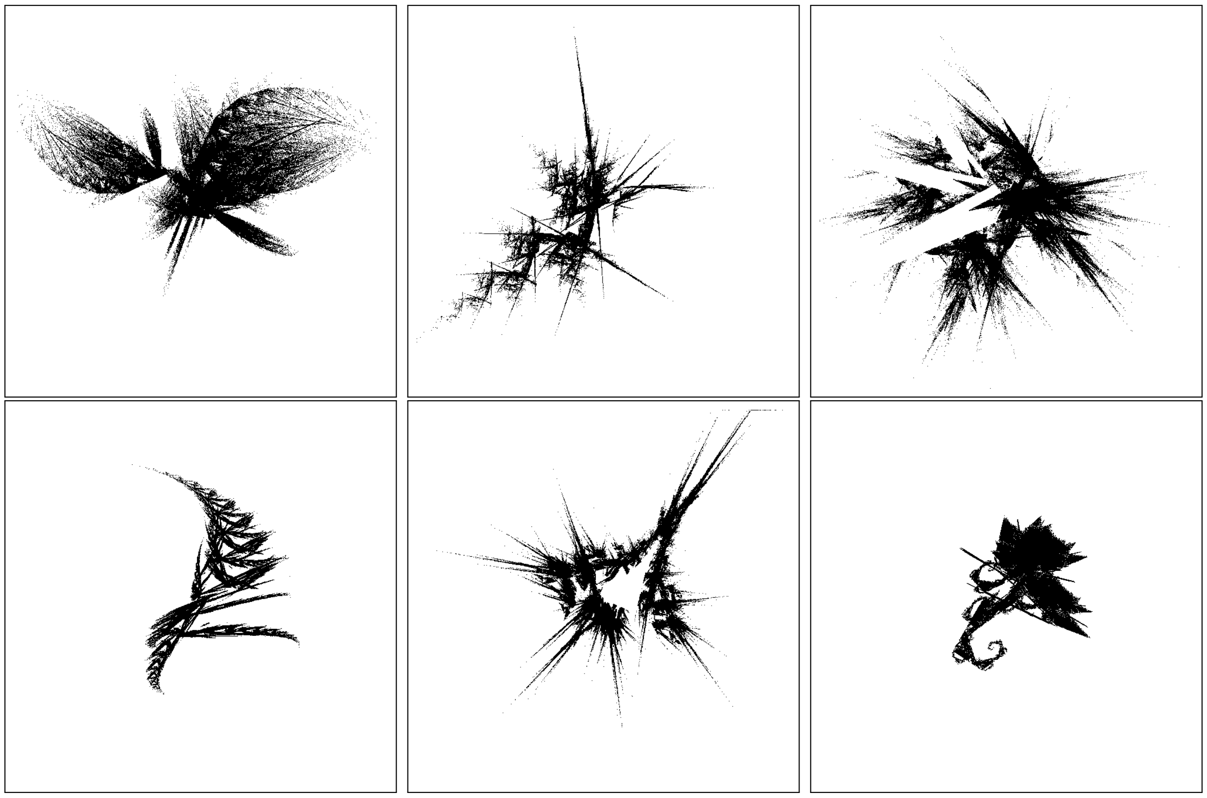


Figure 4.4: Examples of attractors generated from piecewise affine IFSs. The top row is produced with the piecewise condition being whether the x coordinate is above or below zero. The bottom row corresponds to IFSs whose piecewise condition is based on the y coordinate being above or below zero.

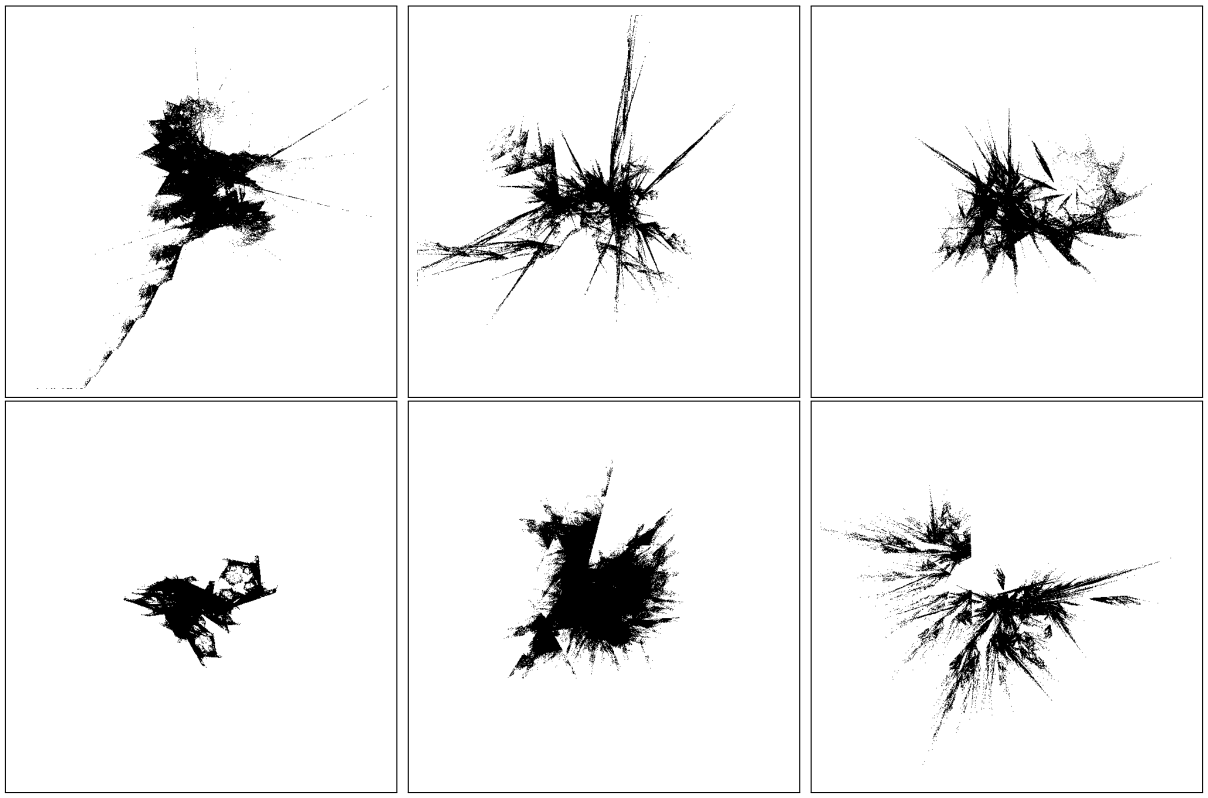


Figure 4.5: Examples of attractors generated from piecewise affine IFSs. The top row is produced with a piecewise boundary defined by $x = 0$ for $y \geq 0$, and $y = 0$ such that $x \geq 0$. The bottom row corresponds to IFSs whose piecewise condition is defined by $x + y = 0$.

attractor switches branches. This is simply because it is only the function's input that is limited to a specific region; each branch of the each function can produce values on both sides of the piecewise boundary. However, many of the fractals generated exhibit distinct, unnatural looking discontinuities. This is shown in several of the attractors displayed in Fig. 4.4, where it appears as though pieces of the fractal are cut off. In addition, these lines do not seem to reflect the orientation of the piecewise boundary, whether it is vertical or horizontal. This also makes sense as each component is dependent on both coordinates of the previous iteration of the random algorithm. With this in mind, what would some attractors look like whose piecewise boundaries depends on both coordinates? Some examples of this are shown in Fig. 4.5.

With these piecewise boundaries depending on both coordinates, we can see that some of the fractals not only possess areas that look cut off, but attributes corresponding to chunks being taken away between two intersecting lines. These can be understood by considering the deterministic algorithm. Suppose our initial set is the square $[-1, 1]^2$, and that the piecewise boundary is given by $x = 0$. Then after the first iteration, one branch of the IFS transforms half the initial set, while the other branch transforms the other half. Though, each branch is not able to produce the same set as if it were not cut off by the piecewise boundary. Hence, some portions of an attractor may only be made through points on the other side of the boundary. These missing sections then get propagated throughout the attractor as the algorithm progresses. These iterations of the deterministic algorithm can be visualized in Fig. 4.6.

There is a problem that arises with this realization, though: Should all transformations of a branch of the IFS produce sets on its own side, we cannot necessarily produce the full attractor. For example, the crystal-like fractal shown in Fig. 2.5 completely fits in the region $[0, 1]^2$. So, if we use a piecewise boundary defined by $x = 0$, and the initial point or set of either generation algorithm has only positive x coordinates, we will only generate the crystal fractal. Additionally, if we are using the random algorithm and the initial point does have a negative x coordinate, as soon as it gets transformed to a point with a positive x coordinate, it will never cross the piecewise boundary again. Therefore, we must ensure the following: Let there be n branches to each piecewise function, each being defined on a region Q_i disjoint from all others. Let A_i be the attractor of the i^{th} branch, and associate each region Q_i with a node, or vertex in a directed graph. Then we can say the node associated with Q_i is connected to the node associated with Q_j if $A_i \cap Q_i$ and $A_i \cap Q_j$ are both nonempty. We then require this corresponding directed graph to be strongly connected to ensure the the random algorithm will utilize each branch of each IFS. Additionally, we can consider piecewise functions that are not defined for disjoint regions by simply splitting the regions

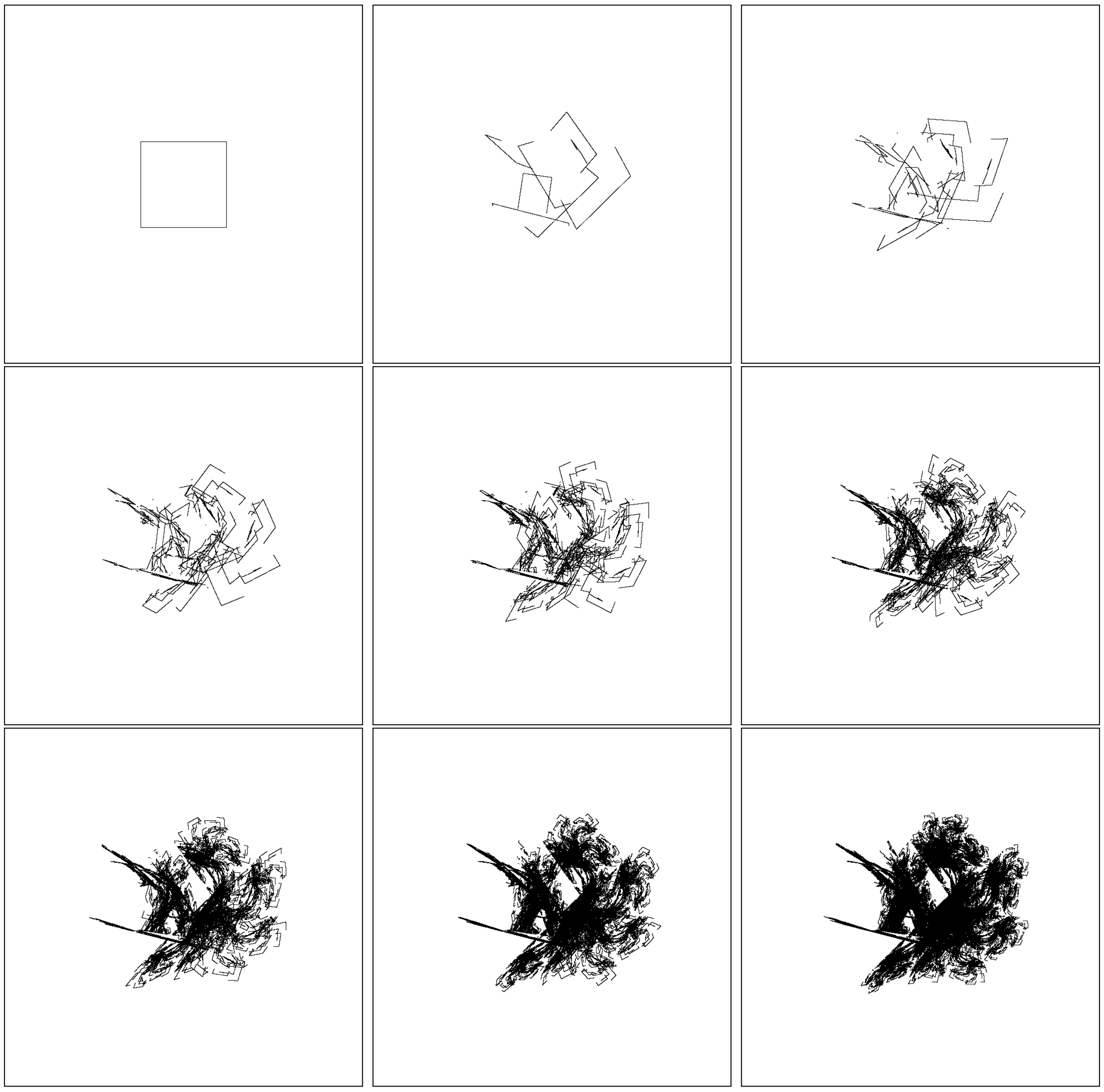


Figure 4.6: Using the deterministic algorithm to generate a piecewise affine IFS in order to display how “missing pieces” of the attractor are created. The piecewise boundary is defined by $x = 0$.

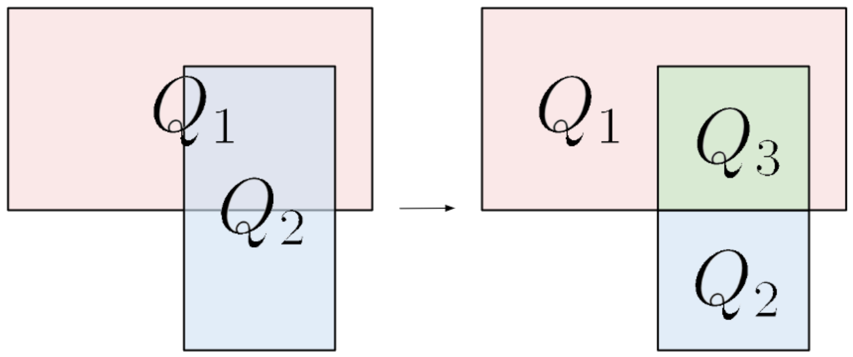


Figure 4.7: An example of how one can make disjoint regions for overlapping piecewise function conditions.

where there is an overlap, as shown in Fig. 4.7. The IFS on the overlapping region can then be the union of the overlapping IFS branches.

Another concept, which may seem like an issue at first, is what happens when one branch of the IFS produces a point on the other side of the piecewise boundary that is not part of the attractor for that side's branch. One may think that this point is not part of the attractor of the piecewise IFS, but this may be incorrect. We can think of the generation of these piecewise IFSs like a strange version of tug of war between the branches of the IFS. The attractor is then the equilibrium state of the game. Consider an attractor corresponding to a piecewise IFS generated with an infinite number of iterations. If we run it through one step of the deterministic algorithm, each branch will take everything on its side and transform it one step closer to its own attractor. Doing this will take away points on each side that do not correspond with their respective attractors. Simultaneously, though, this produces new points on the other side of the piecewise boundary that are exactly the points that were taken away. A simplistic version of this can be visualised in Fig. 4.8. Note that this is actually how regular IFSs work as well. Each point in the set no longer included after the transformation of a given function in the IFS is produced by one of the other functions. These piecewise IFSs are then similar to IFSs of IFSs in the sense that their attractors are

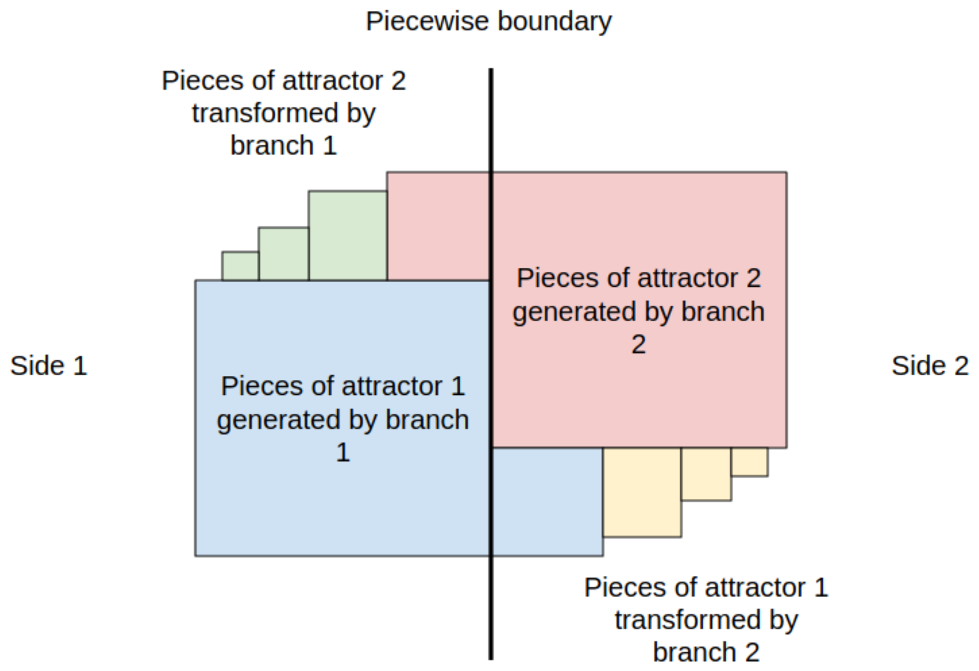

Piecewise boundary
Pieces of attractor 2 transformed by branch 1
Pieces of attractor 2 generated by branch 2
Side 1
Pieces of attractor 1 generated by branch 1
Side 2
Pieces of attractor 1 transformed by branch 2

Figure 4.8: A simplistic depiction of how the attractor of a piecewise IFS incorporates pieces of each branch's attractor.

constructed from pieces of each branch's attractor. Though, we have not yet shown that the random algorithm actually works for these function systems.

Theorem 4.1. *Let W be an IFS with corresponding Hutchinson operator H_W, which is a contraction mapping in $(\mathcal{H}(X), d_{\mathcal{H}})$. Then the points x_n generated by the random algorithm become arbitrarily close to the attractor, A, of W. Additionally, for any $\epsilon > 0$, $\exists N$ such that if $n > N$, then $d_{\mathcal{H}}(x_n, A) < \epsilon$.*

Proof. Let c be the contraction factor of H_W. Then the determinstic algorithm produces the sequence of sets $\{S_n\}_{n=0}^{\infty}$, where $S_n = H_W(S_{n-1})$. Their distance to A is then

$$\begin{aligned} d_{\mathcal{H}}(S_n,\, A) &= d_{\mathcal{H}}(H_W(S_{n-1}),\, H_W(A)) \\ &\leq c\, d_{\mathcal{H}}(S_{n-1},\, A) \\ &\leq c^n\, d_{\mathcal{H}}(S_0,\, A). \end{aligned}$$

Let $\{x_n\}_{n=1}^{\infty}$ be the sequence of points generated by the random algorithm such that $x_n = w_{\sigma_n}(x_{n-1})$, where σ_n is a random index, and $w_i \in W$. Then we see $x_n \in S_n$. In other words, the sequence generated from the random algorithm is such that the n^{th} point lies in the set produced from the n^{th} application of the Hutchinson operator. This can be seen as a modified version of the setup to the nested interval theorem. Now let S_n be the sequence of sets generated from the deterministic algorithm such that S_0 is the singleton set $\{x_0\}$. Then the distance between the n^{th} point in the random algorithm and the attractor is found to be

$$\begin{aligned} d_{X,point}(x_n,\, A) &\leq d_{\mathcal{H}}(\{x_n\},\, A) \\ &\leq \max\{d_{\mathcal{H}}(\{x_i\},\, A) \,|\, x_i \in S_n\} \\ &= d_{\mathcal{H}}(S_n,\, A) \\ &\leq c^n\, d_{\mathcal{H}}(\{x_0\},\, A). \end{aligned}$$

$\square$

Closed piecewise function boundaries are another interesting case to consider. Some examples of this are shown in Fig. 4.9, where the piecewise boundaries are defined by $|x| + |y| = 1$ and $x^2 + y^2 = 1$, for the top and bottom rows, respectively. We can see from these images that the fractals appear to be partially made from pieces of squares and circles. We get both cut out chunks of these shapes as well as filled in squares and circles. The cut out

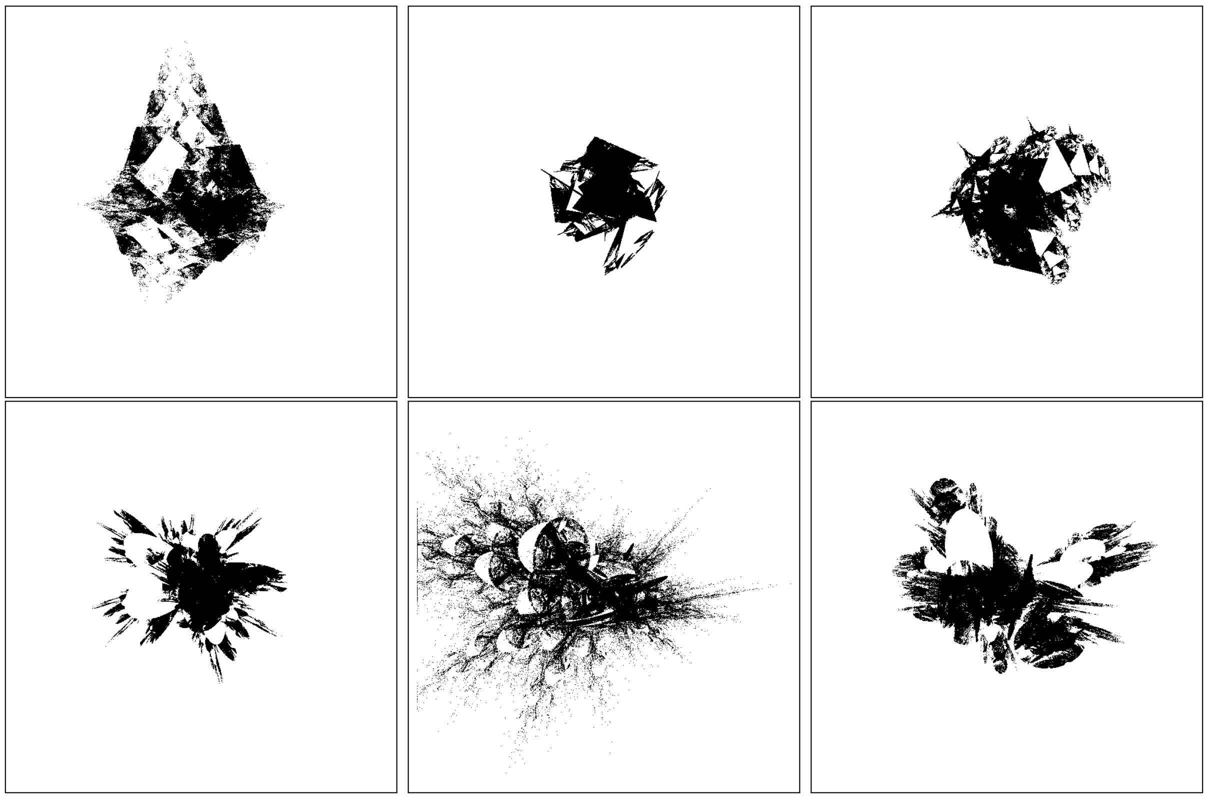


Figure 4.9: Examples of attractors generated from piecewise affine IFSs. The top row is produced with the piecewise boundary defined by $|x| + |y| = 1$. The bottom row corresponds to IFSs whose piecewise boundary is given by $x^2 + y^2 = 1$.

chunks can be explained similar to before; affine mapping being applied to everywhere but a given region, resulting in missing shapes from the attractor. Similarly, the filled in squares and circles can be explained as being mappings from the branch inside the shape, to areas on the other side of the piecewise boundary. This means that we can see which branch of the IFS formed specific parts of the attractor; the filled in shapes are created from the branch inside the shape, while the hollow shapes are created from the branch outside of it.

Fractal Splicing

One concept we can apply with these piecewise IFSs is **fractal splicing**: Combining two, or more IFSs by making each one a branch of a piecewise IFS. Since we know the attractor of the piecewise IFS is constructed from pieces of each branch's attractor, we may be able to obtain features of both. We will restrict ourselves to only splicing two IFSs together. There are a few things to keep in mind before we do this, though. Firstly, we can, for the most part, scale these fractals. By scaling the attractors of each branch, the majority of the piecewise IFS attractor will be in the same region. The only components of the attractor that may not be within the desired region are transformed pieces of the other branch's attractor (the green and yellow portions shown in Fig. 4.8). This allows us to combine a wider variety of fractals due to the previously mentioned condition that the corresponding directed graph must be strongly connected. Results of splicing some common fractals is shown in Fig. 4.10. Note that the number of functions for each IFS combined in the bottom two rows is not equal. In these cases, we only made as many functions piecewise as there were in the IFS possessing less functions. The remaining functions of the original IFS with a greater number of them, were then kept purely affine. This means that one can obtain different results based on which functions were made to be piecewise when splicing two fractals with an unequal amount of functions.

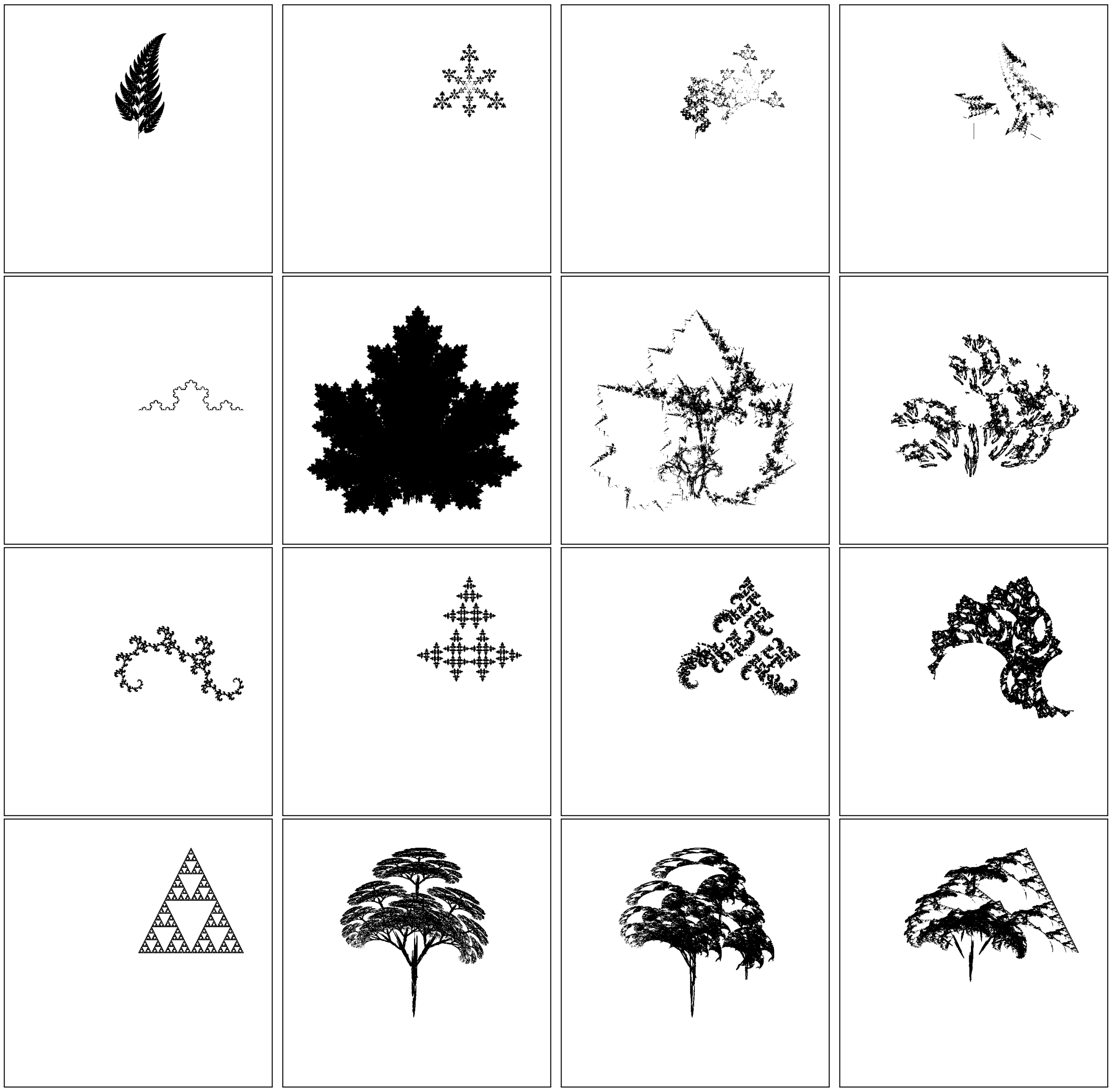


Figure 4.10: Examples of fractal splicing. The attractors in the left two columns were spliced together to make the fractals on the right. For each of the spliced fractals the piecewise boundary is given by $x^2 + y^2 = 0.25$. The left column of spliced fractals has the IFS of the attractor in the left column being defined inside the circle boundary, whereas in the right column, the IFS branches are switched.

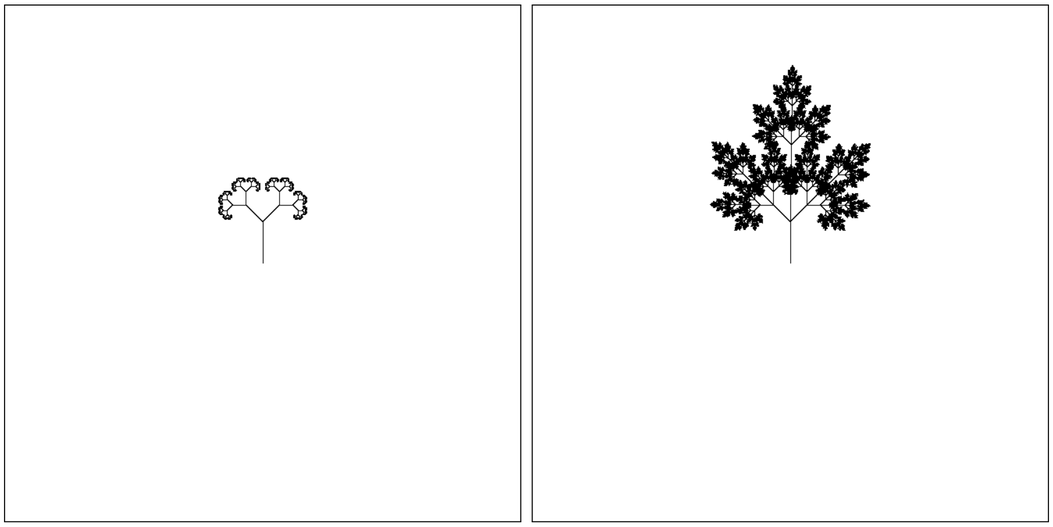


Figure 4.11: A depiction of how introducing a new mapping from another IFS can incorporate features of that IFS in the attractor. A mapping from the maple leaf fractal was added to the IFS pertaining to the attractor on the left. The fractal produced from the new IFS is shown on the right.

We can see from these examples that by splicing IFSs together, the attractor takes on attributes of both fractals from which it was made. One interesting thing to note about this, is that one typically gets “better” results when the number of functions is not equal. What is meant by this, is that attributes from the attractor of each branch were more easily seen, and the resulting fractal is typically more natural looking, as opposed to pieces in arbitrary locations, such as in the top row of Fig. 4.10. This makes sense when considering what would happen in an iteration of the deterministic algorithm. Having at least one function in the IFS that is not piecewise will allow it to act on the entire set of points, and blend the features of each branch. Another way that we can think of this is maintaining a fully piecewise IFS, but adding a sufficient number of contraction mappings to the IFS consisting of fewer functions until both branches have the same amount. Simply including an extra contraction mapping by itself can introduce features of the other attractor. This can be seen in Fig. 4.11 where a single mapping from the leaf fractal was added to the IFS of the attractor shown on the left, producing an IFS with the attractor on the right.

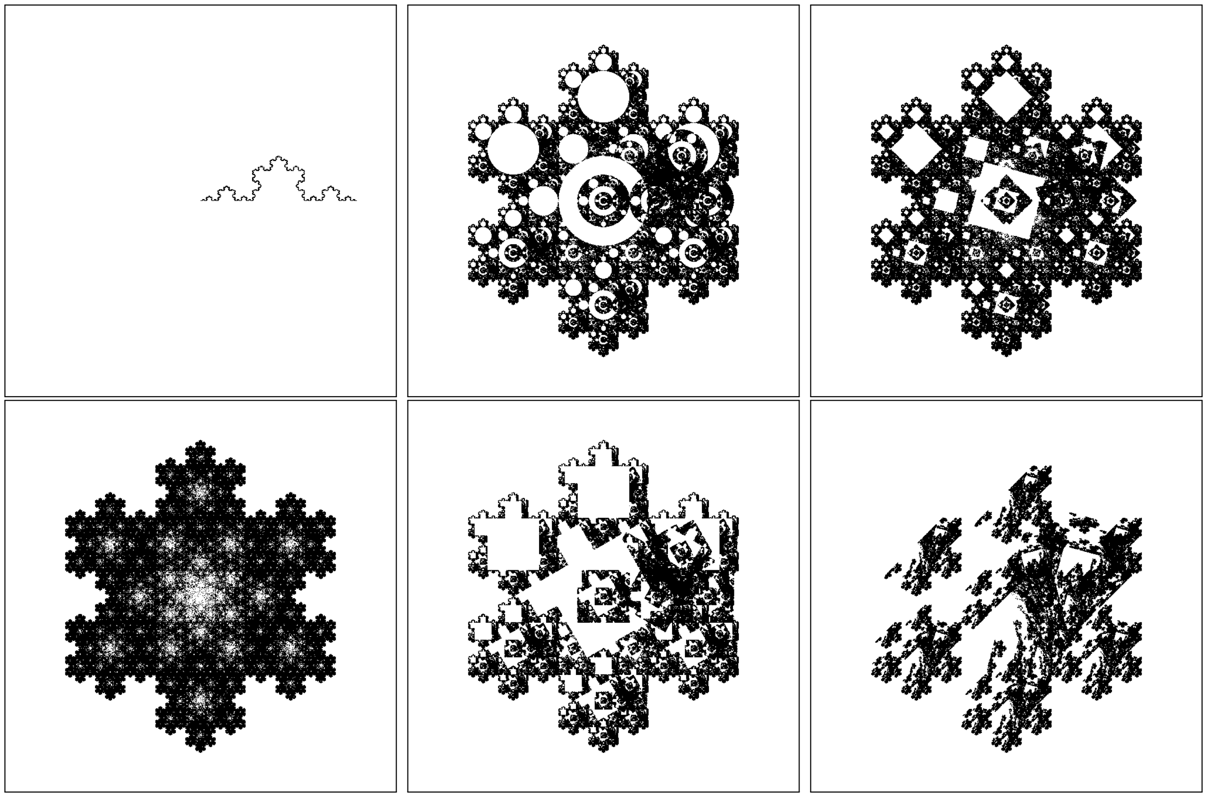


Figure 4.12: The effects of changing the piecewise boundary when splicing two attractors together. The IFSs corresponding to the attractors in the left column were spliced together to create the rest of the images. The other images in the top row correspond to piecewise boundaries being defined as $x^2 + y^2 = 0.25$, and $|x| + |y| < 0.5$ going from left to right respectively. Similarly, the bottom two spliced fractals have boundaries of $x = 0$ for $y \geq 0$ and $y = 0$ for $x \geq 0$, and $y = x$ respectively.

Another thing to keep in mind when splicing fractals together is how the piecewise boundary will manifest itself. As we saw previously with circles and squares being part of the attractor, the piecewise boundary can substantially change how the spliced fractal looks. This can be seen in Fig. 4.12 where we splice together the IFSs corresponding to the attractors in the left column to create the rest of the attractors with various piecewise boundaries.

Chapter 5

Deep Learning Background

Recent breakthroughs in machine learning, those algorithms that improve through experience, have provided us with many new and useful technologies. From programs that can convert hand written notes into text on a computer, to ones that can describe the setting within an image, these algorithms have a wide array of applications [8]. The purpose of this chapter is to discuss some of the theory and common practices of particular machine learning areas so that we can apply them to the fractal databases in order to predict parameters for a given attractor in the subsequent chapter.

5.1 Feedforward Neural Networks

A feedforward neural network is one of the most basic and essential concepts of deep learning; a soon to be defined subfield of machine learning. These networks, also called multilayer perceptrons (MLPs), are typically used as classifiers. That is, the purpose of the network is to take in data, and classify each piece into a known category. One way that the network can classify data is by outputting a vector where each component represents a specific category. Then by convention, the largest component of the output vector is the category the network chooses for the given piece of input data [8]. We can begin to express this mathematically by defining our goal function as $\boldsymbol{y} = f^*(\boldsymbol{x})$, which maps input data $\boldsymbol{x}$ to the output vector $\boldsymbol{y}$. Then our feedforward neural network is expressed by $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are parameter values of the network which are learned to best approximate the function $f^*(\boldsymbol{x})$.

Although this concept may be easy to understand, it may still be elusive as to why this is called a feedforward neural network. There is a purpose to this terminology, though. The function $f(\boldsymbol{x}; \boldsymbol{\theta})$ is typically constructed from multiple other functions. This is done for a few reasons. As mentioned above, the input and output of the network are vectors, hence each function represents a different transformation on the vector from the previous function output. Splitting up the network into these separate functions not only makes it easier to understand what these transformations are doing, but allows us to apply other operations, such as regularization techniques, which will be discussed in a later section, at multiple areas throughout the network. Suppose our network contains the functions f_1, f_2, f_3, and f_4. Then we might construct our feedforward neural network in a chain as $f(\boldsymbol{x}; \boldsymbol{\theta}) = f_4(f_3(f_2(f_1(\boldsymbol{x}))))$, where each function contains a portion of the parameters $\boldsymbol{\theta}$. This is where the term feedforward is derived; information is fed from one function directly into the next. If any output were to be fed back into itself, or a previous function, we would no longer have a feedfoward network, but instead a recurrent neural network. Recurrent neural networks are the primary form of machine learning used in continuous input applications such as speech recognition, and natural language processing [8].

A graph may be constructed that depicts which components of a vector get mapped to another, and in what combinations, giving the network portion of the term feedforward neural network. Lastly, a note on the ”neural” terminology used in this field, which is inspired from biology: We can consider each component of one of the vector valued functions to be a neuron, or node that takes in a value and outputs another. The computation of that node along with the network of connections can then be thought of like a brain.

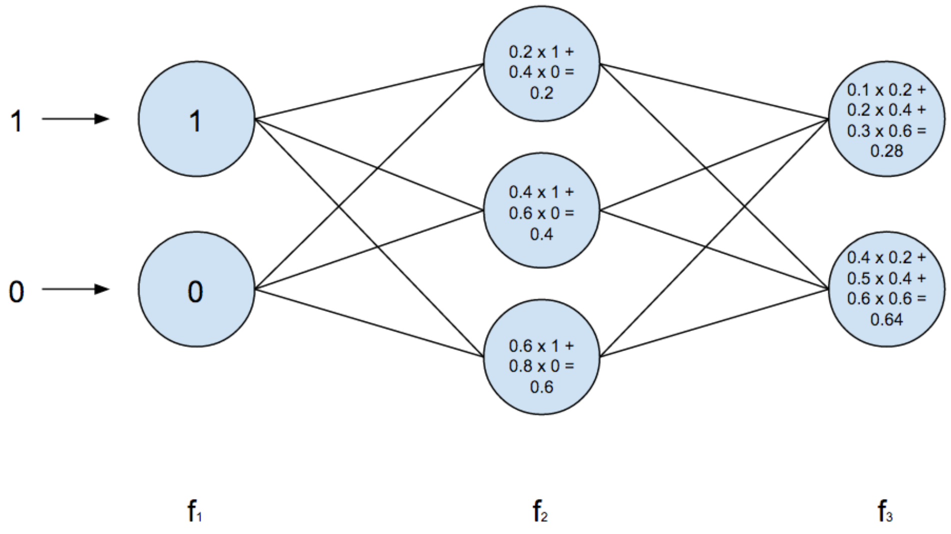

1
0
1
0
0.2 x 1 + 0.4 x 0 = 0.2
0.4 x 1 + 0.6 x 0 = 0.4
0.6 x 1 + 0.8 x 0 = 0.6
0.1 x 0.2 + 0.2 x 0.4 + 0.3 x 0.6 = 0.28
0.4 x 0.2 + 0.5 x 0.4 + 0.6 x 0.6 = 0.64
f₁
f₂
f₃

Figure 5.1: A graphical representation of the simple neural network in Ex. 5.1

Example 5.1. *In this example we demonstrate how data is fed forward through a simple neural network, and how it calculates all the intermediate and output values. The layout of the network is shown in Fig. 5.1 where the functions f_1, f_2, and f_3 are given by*

$$\begin{cases} f_1(x,\, y) & = (x, y) \\ f_2(x,\, y) & = (0.2x + 0.4y,\, 0.4x + 0.6y,\, 0.6x + 0.8y) \\ f_3(x,\, y,\, z) & = (0.1x + 0.2y + 0.3z,\, 0.4x + 0.5y + 0.6z) \end{cases}$$

Note that since this is a feedforward neural network, the dimension of the output of one layer must match the dimension of the input of the layer it feeds into.

In Fig. 5.1, each blue circle represents a node, or one of the components of the vector valued functions, and each line represents how that component is connected to the others. We are then approximating some function $f^*(x,\, y)$ with our network of the form $f(x,\, y;\, \boldsymbol{\theta}) = f_3(f_2(f_1(x,\, y)))$ where the parameter values $\boldsymbol{\theta}$ is the set of all coefficients in each of the functions. For this example, we give the network the values $(x,\, y) = (1,\, 0)$ and obtain an *output of* (0.28, 0.64)*. Note that the output does not have to be, and typically is not of the same dimension as the input vector.*

Now that we have a basic understanding of what a feedforward neural network is, we can go over some more terminology in order to describe different networks efficiently.

Definition 5.1 (Layers)**.** *Each vector valued function, which is represented by a vertical line of nodes in Fig. 5.1, is called a layer. The first layer in the network is called the input layer, and the last is called the output layer. All layers in between the input and output layers are called hidden layers. If it were not for the explicit calculations shown in Fig. 5.1, we would not know what values the nodes in the hidden layers are storing.*

Definition 5.2 (Depth)**.** *The number of layers used to construct a neural network is called the depth of the model.*

Definition 5.3 (Width)**.** *The number of nodes in a given layer is called the width of that layer.*

Definition 5.4 (Fully Connected)**.** *When each node in one layer of a network is connected to each node of the next layer, it is called fully connected. If each layer of a network possesses this property, it is called a fully connected neural network.*

Using these definitions, we can now state that the network shown in Fig. 5.1 is fully connected with a depth of three, and it possesses one hidden layer of width three. Typically, the more complicated the function you are trying to approximate, the deeper the model needs to be; this is where the term deep learning comes from [8]. Creating networks that contain very wide layers often leads to overfitting. This is akin to attempting to model data with a polynomial of a degree too high that it no longer generalizes well. This problem will be addressed in the regularization section.

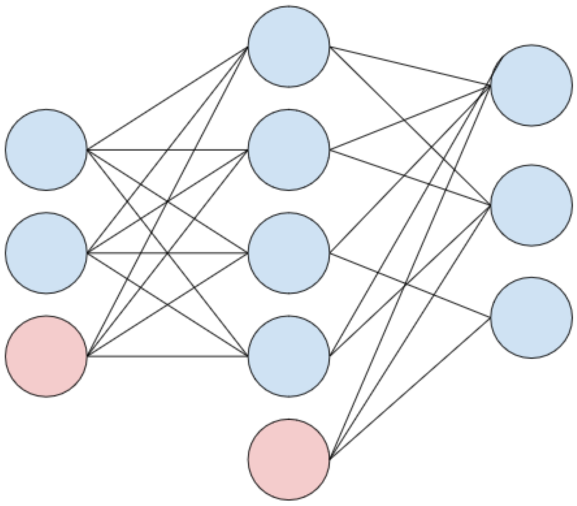


Figure 5.2: An example of a neural network that contains bias nodes. The blue circles represents the regular nodes whereas the red ones represent the bias nodes.

There is another way to think about our network that may be more intuitive and favor its graphical representation. All the coefficients for the linear transformations in each node can be thought of as a weight corresponding to each connection. The weights need not be normalized, but represent the strength of the connection between the two nodes. The value in each neuron is then given by the sum of the values in the previous layer multiplied by the corresponding weights.

As of now, we have only looked at linear transformations, however, we can improve the capacity for our networks to approximate by expanding this to affine transformations. Affine mappings can be constructed by including another node that is independent of the inputs, and has a corresponding weight to each node in the next layer. The parameters of this node, termed a **bias node**, will also be included in the total set of parameters, $\boldsymbol{\theta}$. An example of a network containing bias nodes is given in Fig. 5.2. Note that the bias nodes are not included in the width of the layers. So, this network has a depth of three, a single fully connected hidden layer of width four, and a partially connected output layer of width three.

With this framework, we can represent the computation in a given layer by

$$f(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{b}) = \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} + \boldsymbol{b}.$$

In this equation, $\boldsymbol{w}$ is an $m \times n$ matrix of weights with m and n being the number of nodes in the current and previous layers, respectively, $\boldsymbol{x}$ is a vector of the outputs from the previous layer, and $\boldsymbol{b}$ is the vector of biases. Note that we can represent missing connections in the weight matrix with a value of zero. Additionally, the way the bias nodes are actually implemented is through setting them to one, and multiplying them too by a weight value. This results in the transformation

$$f(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^{\mathsf{T}} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix},$$

however, we will continue with the original notation for simplicity. We can further increase the capacity for our network to approximate functions by introducing non-linear transformations, the topic of the next section.

5.2 Activation Functions

With a chain of affine functions, there is a limit on the transformations a given network can approximate well. Complicated nonlinear functions would be exceedingly difficult to approximate. Hence, we introduce a nonlinear transformation to each layer called the activation function.

Theorem 5.1 (Universal Approximation Theorem). *We say that a set of functions $F \in L^{\infty}_{loc}(\mathbb{R}^n)$ is dense in $C(\mathbb{R}^n)$ if for every function $g \in C(\mathbb{R}^n)$ and compact set $K \subset \mathbb{R}^n$, there* *exists a sequence of functions $f_i \in F$ such that*

$$\lim_{i \to \infty} ||g - f_i||_{L^\infty(K)} = 0.$$

Let M denote the set of functions in $L^\infty_{loc}(\mathbb{R}^n)$ such that the closure of the set of any of the function's discontinuities is of Lebesgue measure zero. Then, if $\phi \in M$, we define

$$\Sigma_n = span\{\phi(\boldsymbol{w}^\top \boldsymbol{x} + b) : \boldsymbol{w} \in \mathbb{R}^n,\, b \in \mathbb{R}\}.$$

Then Σ_n is dense in $C(\mathbb{R}^n)$ if and only if ϕ is not an algebraic polynomial almost everywhere.

Proof. For the proof of this theorem, see [13]. □

What the universal approximation theorem is saying, is that so long as we have one hidden layer with a locally bounded, piecewise continuous, nonpolynomial transformation, we can approximate any function to any degree that we wish [8]. The caveat here is that it does not specify how wide the hidden layer must be in order to achieve the desired accuracy.

The first activation function that we will examine is called the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

This function was first introduced as it possesses some nice properties: The sigmoid function satisfies all conditions required by the universal approximation theorem, and its derivative can be expressed in terms of itself as $\sigma(x)(1 - \sigma(x))$, which will significantly reduce the number of computations needed when training the model. Additionally, the range of the function is (0, 1), which allows us to think of outputs more like probabilities. As a result of this range, it enables the network to deal with a wider array of inputs, as they will all be contained within that interval. The sigmoid function and its derivative is shown in Fig. 5.3.

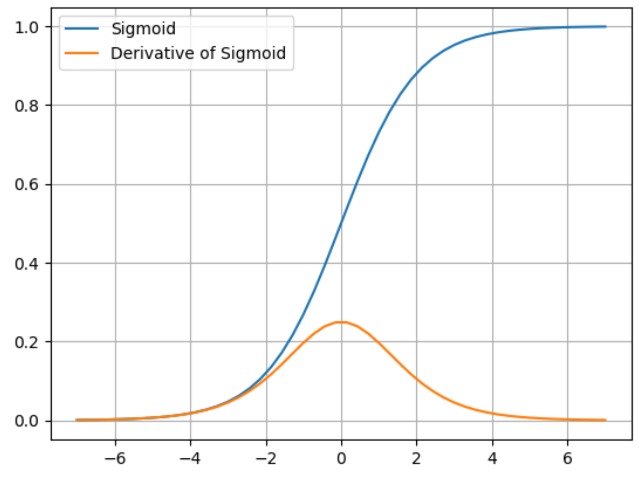


Figure 5.3: A plot of the sigmoid function as well as its derivative.

Example 5.2. *In this example we will approximate the Boolean XOR (eXclusive OR) function using a simple feedforward neural network that contains biases, and uses the sigmoid activation function. Boolean functions take two input parameters that are either true or false, represented by one and zero respectively. The XOR function returns true only if one of the inputs is true while the other is false. The network that we will use to approximate this function has the architecture shown in Fig. 5.4, where the functions it represents are defined as*

$$\begin{cases} f_1(x, y) &= \begin{bmatrix} x & y \end{bmatrix}^\mathsf{T} \\ f_2(x,y) &= \sigma\left(\boldsymbol{w}_2 \begin{bmatrix} x & y \end{bmatrix}^\mathsf{T} + \boldsymbol{b}_2\right) \\ f_3(x, y, z) &= \begin{bmatrix} 0.7818 & 2.5948 & 2.4807 \end{bmatrix} \begin{bmatrix} x & y & z \end{bmatrix}^\mathsf{T} + \begin{bmatrix} -1.4811 \end{bmatrix} \end{cases}$$

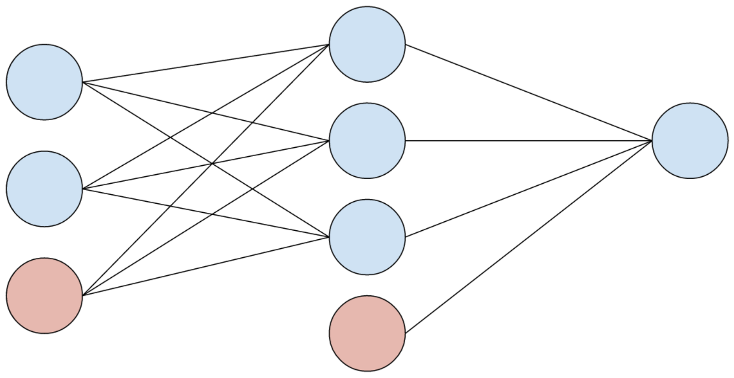


Figure 5.4: A visual representation of the neural network used in Ex. 5.2 to model the XOR function. The red circles are the biases, and the blue are regular nodes.

Input	True Value	Model Output
(0, 0)	0	-5.9605×10^{-6}
(0, 1)	1	1
(1, 0)	1	1
(1, 1)	0	-6.9141×10^{-6}

Table 5.1: This table displays the input, output, and true values for the XOR neural network model of Ex. 5.2.

where the matrix of weights $\boldsymbol{w}_2$, and vector of biases $\boldsymbol{b}_2$ are given by

$$\boldsymbol{w}_2 = \begin{bmatrix} 2.1851 & 2.3421 \\ 2.4837 & -3.0120 \\ -2.1454 & 2.532 \end{bmatrix} \qquad \boldsymbol{b}_2 = \begin{bmatrix} 2.21212 \\ -1.9324 \\ -1.4939 \end{bmatrix}$$

Output from the model can be seen in Table 5.1. Of course, the model only came to be able to produce such accurate output, and have the parameters it does after it was trained to do so, a topic of a later section. However, the point of this example is to demonstrate that we can model the XOR function quite accurately by incorporating the sigmoid activation function.

One should only use the sigmoid activation function for shallow neural networks, though. There is an issue that arises with it in deep neural networks that will be discussed with the proper background in the gradient descent and optimization section. So, we turn to another common activation function called the Rectified Linear Unit (ReLU). ReLU is defined as,

$$ReLU(x) = \max(0,\, x).$$

The derivative of the ReLU function is defined everywhere except at $x = 0$ where a value of zero is assigned. One may notice that the ReLU function does not satisfy the requirements of the universal approximation theorem. However, the ReLU activation function has been shown to be a universal approximator as well.

Theorem 5.2. *Let f be a Lebesgue integrable function from $\mathbb{R}^n$ to $\mathbb{R}$. Then, a fully connected network with n input nodes, the ReLU activation function, and a hidden layer of width $n + 4$ can approximate f to arbitrary accuracy. Additionally, except for a negligable set of functions, f cannot be approximated if the width of the hidden layer is no more than n.*

Proof. For the proof of this theorem, see [14]. □

Since the beginning of its use, ReLU has become a multipurpose activation function; it is used as the default for many neural networks. One problem that comes with it, though, is the possibility of "dead nodes". As will be seen in the gradient descent section, should a node in the network return a negative value, when paired with the ReLU function, it will not be able to receive updates [8]. If this were to continue happening no matter the input, say if the network learns a large negative bias, then the node representing this transformation is considered "dead".

Because of the possibility of dead nodes with the ReLU activation function, it is recommended to initialize all biases with a small positive value. Additionally, many variants of the ReLU function have been made to overcome this issue, and have been shown to perform just as well [8]. These variants include

- Leaky ReLU: $\max(0.1x,\, x)$
- Softplus: $\ln(1 + e^x)$
- PReLU: $\max(\alpha x,\, x)$, α is another learnable parameter

and many others. These functions attempt to overcome the problem of dead nodes by introducing a nonzero derivative when the input of the function is negative.

5.3 Loss Functions

Gradient descent is the fundamental mechanism behind deep learning. It is the method by which these neural networks learn their parameters, and the reason that they work at all. Using outputs that we know or want for a given input, and using the functions covered in this section, we can calculate an error, or loss, which is to be minimized through gradient descent. **Supervised learning**, the type of machine learning that will be used throughout this thesis, goes as follows:

- Data is fed forward through the network until the outputs are obtained
- A loss, or cost is calculated using the model outputs and desired outputs
- The gradient of the loss function with respect to each of the parameters of the network is calculated
- The calculated gradients are used to update each of the parameters of the network
- All steps are repeated until a sufficiently low loss is achieved, or stopping criteria are met

In this section we cover various loss functions, and appropriate times to use them. Choosing the correct loss function can drastically decrease the amount of time required to train a given neural network, and there are many different loss functions to choose from. All loss functions, however, need to be differentiable everywhere, as we will be applying gradient descent.

Definition 5.5 (L1 Loss)**.** *Let $\hat{\boldsymbol{y}}$ be the output vector of length n from a neural network, and let $\boldsymbol{y}$ be the true values. Then, the L1 loss can be calculated as*

$$L1(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n} |\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i|$$

Note that this function is not differentiable on any of the lines $\hat{\boldsymbol{y}}_i = \boldsymbol{y}_i$. To account for this the partial derivative with respect to $\hat{\boldsymbol{y}}_i$ at such points is defined to be zero.

Definition 5.6 (L2 or Mean Squared Error Loss)**.** *Let $\hat{\boldsymbol{y}}$ be the output vector of length n from a neural network, and let $\boldsymbol{y}$ be the true values. Then we define the L2, or Mean Squared Error (MSE) loss to be*

$$MSE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2$$

Both of these simpler loss functions can be applied to any neural network in general. However, distinguishing which loss function would work best depends greatly on the data and purpose of the network. For example, if your data had a significant amount of outliers, it would be better to use the L1 loss, as L2 would likely produce larger losses, and result in a gradient that does not necessarily reflect the dataset due to those outliers.

Definition 5.7 (Support Vector Machine)**.** *A neural network used to construct one, or more hyperplanes with the purpose of classifying data by means of maximizing the margin between the data and hyperplane is called a support vector machine (SVM).*

Definition 5.8 (Hinge Loss). *Let y be the output from a neural network. Then the hinge loss is calculated as*

$$L_{hinge}(y) = \max(0,\, 1 - t \cdot y),$$

where t is a scaling factor defining a threshold of $1/t$ for the desired output. Typically t is chosen to be ± 1, where the sign is always chosen to match that of the true category.

The hinge loss is a more specialized loss function often used for SVMs. If a given piece of data is on the correct side of the margin, the network receives a loss of zero. However, if a given piece of data is on the wrong side of the margin, the network would receive a loss proportional to its distance from the margin.

Example 5.3. *To get a sense of how this loss function works, suppose we have a classification network that takes in pixel values of an image and has one output node: We would like it to return a positive number if the image is of a dog, and a negative number if the image is of a cat. This is a binary classification network. Now consider the following cases where in the hinge loss function we are using has a scaling factor of $t = 1$ if the image is truly a dog, and $t = -1$ for images of cats:*

1. *The network returns a value of* 5.0
2. *The network returns a value of* -0.3
3. *The network returns a value of* 0.0

In the first case, the neural network is indicating that the image is of a dog. We can also tell that this is a relatively strong prediction as 5.0 *is past the threshold of the hinge loss by a factor of five. So if the input image truly was a dog, then the hinge loss returns a value of zero. However, if it were really a cat, then the hinge loss would return* $\max(0,\, 1 - (-1)(5)) = 6$.

Suppose that the image is a cat in the second case. While the network predicts the correct category by returning a negative value, it fails to meet the threshold and hence we still have a non-zero loss of $\max(0, 1-(-1)(-0.3)) = 0.7$.

For that last case, it does not actually matter what the image is, as the output is an equal distance from the threshold for both cats and dogs. Although the chances of this happening in reality are quite small, it would recieve a loss of one.

If our dataset has more than two categories, we cannot use a binary classifier like in the above example. Instead, as mentioned previously, we can create a network with multiple output nodes where each one corresponds to a specific category. The output node with the largest value is then taken to be the model's prediction.

Definition 5.9 (Softmax Function)**.** *Let $\hat{\boldsymbol{y}}$ be the output vector of length n from a neural network. Then, the softmax function takes in a single component of $\hat{\boldsymbol{y}}$, and computes the transformation*

$$Softmax(\hat{\boldsymbol{y}}_i) = \frac{e^{\hat{\boldsymbol{y}}_i}}{\sum_{j=1}^{n} e^{\hat{\boldsymbol{y}}_j}}.$$

Note that this is not a loss function as it does not return a single value for the network. Instead, it transforms the outputs to a more useful form.

The softmax function can be very useful for classification models as it provides the outputs with some very useful properties. Namely, the normalization of the outputs allows us to treat them like probabilities. Then for any given data, we receive an array of probabilities of the possible categories. In addition, the exponentiation within the function provides a useful property. Not only does this make the function an everywhere differentiable variant of the argmax function, which simply takes the largest input, but the exponentials make it much more difficult to achieve zero loss. In order to obtain a loss value of zero, the correct output node would have to be infinite. Then, similar to the hinge loss, even if the model predicts the correct category, it will continue receiving non-zero loss values so that its parameters can be updated in a way to produce stronger predictions, that is, one category has a much higher probability than the others.

Definition 5.10 (Negative Log-Likelihood Loss)**.** *Let $\hat{\boldsymbol{y}}$ be the output vector of length n from a classification neural network, and $\bar{\boldsymbol{y}}$ be a transformation of $\hat{\boldsymbol{y}}$ such that $\bar{\boldsymbol{y}}_i \in (0,1]$, and the sum of all $\bar{\boldsymbol{y}}_i$ is one. Let $\bar{\boldsymbol{y}}_{class}$ be the element corresponding to the true category of the input data. The Negative Log-Likelihood (NLL) loss is then given by*

$$NLL(\bar{\boldsymbol{y}}) = -\ln\left(\bar{\boldsymbol{y}}_{class}\right).$$

Note that this loss function can only be used on outputs in $(0,1]$, otherwise, this function could result in negative loss values.

The softmax function is typically paired with the NLL loss. Similar to the L2 loss, it penalizes incorrect outputs exponentially more the further the model is from the true values. The combination of softmax with the NLL loss forms the Cross Entropy (CE) loss function.

Definition 5.11 (Cross Entropy Loss)**.** *Let $\hat{\boldsymbol{y}}$ be the output vector of length n from a classification neural network, where $\hat{\boldsymbol{y}}_{class}$ corresponds to the true category of the input data. The Cross Entropy loss is then calculated as*

$$CE(\hat{\boldsymbol{y}}) = -\ln\left(\frac{e^{\hat{\boldsymbol{y}}_{class}}}{\sum_{j=1}^{n} e^{\hat{\boldsymbol{y}}_j}}\right).$$

The CE loss function gets its name from information theory, where the cross entropy quantifies the difference between two probability distributions, p and q, over the same set of events. Specifically, the cross entropy returns the average number of bits needed to identify an event should a coding scheme used to identify an event in the true distribution p, be optimized for the estimated distribution q. For non-trivial distributions the cross entropy is given by

$$H(p,\, q) = -\sum_{x} p(x) \ln\left(q(x)\right),$$

where x goes over all events in the probability distributions. For classification algorithms, however, we know the category of the input, and hence the true distribution, p, collapses to be one in the correct event, and zero for all others. This cancels the sum, resulting in the above formula for the CE loss when accounting for softmax.

Example 5.4. *Suppose the network in Ex. 5.1 was a classification model where the top node indicates that the input is authentic, and the bottom node indicates that the input is a fake. After applying the softmax function to the outputs of* $(0.28,\, 0.64)$*, the new normalized outputs are*

$$(0.28,\, 0.64) \to \left(\frac{e^{0.28}}{e^{0.28} + e^{0.64}},\, \frac{e^{0.64}}{e^{0.28} + e^{0.64}}\right) = (0.411,\, 0.589).$$

Hence the network is indicating that the input is fake with a probability of 58.9%*. If the input were actually authentic, then the CE loss would be*

$$CE(0.28) = -\ln\left(0.411\right) = 0.889.$$

Note that this is in part how generative adversarial networks work; a type of network that takes in an array, and produces outputs such as images of people's faces. One network is designed such that an input image is reduced to a relatively small vector and then expanded once again to the size of the input. Supervised learning is then used to train this network to reproduce its input. After a desired amount of steps has been performed, the part of the network that compresses it to a vector is discarded. Meanwhile, supervised learning is used to train a second network to predict whether an image is authentic, and portrays a real person's face, or if it is a computationally generated face. To obtain the computationally generated *images, random input vectors are given to the image generating network. The output of the authentication network is used to further update the parameters of the image generating network, and whether or not that output is correct is used to improve the capabilities of the authentication network. So, these networks compete against one another; the image generating network is attempting to fool the authentication network. Once the image generating network is able to consistently win this competition, the outputs can be quite realistic!*

5.4 Gradient Descent and Optimization Algorithms

Now that we know how to calculate the error for the output of a neural network, we can begin trying to minimize it using gradient descent. Since the gradient of a function points in the direction of steepest ascent, the method of gradient descent is the act of moving a given function's variables in the opposite direction of the gradient in order to minimize it. More specifically for neural networks, if we have a loss function that tells us how well our model performs, we can treat the parameters of our model as variables, and modify them in the direction opposing that of their respective gradients; the direction of steepest descent.

Example 5.5. *In this example we will perform linear regression on a small dataset using gradient descent. We will model the values shown in Table 5.2 with the function $f(x) = ax+b$. Gradient descent will allow us to find near optimal values for a and b by minimizing the MSE loss function. Initializing both a and b to zero, we obtain an initial loss of*

$$Loss = \frac{1}{11}\sum_{i=0}^{11}(f(x_i) - y_i)^2 \approx 41.75$$

Then, the gradient of this loss function with respect to a is given by

$$\frac{\partial Loss}{\partial a} = \frac{1}{11}\sum_{i=0}^{11}\left[2 \cdot (f(x_i) - y_i) \cdot x_i\right].$$

x_i	0	1	2	3	4	5	6	7	8	9	10
y_i	1.5	1.7	2.4	4.2	5.1	5.8	6.4	7.2	8.7	9.5	10.6

Table 5.2: Data used in Ex. 5.5 for linear regression by means of gradient descent.

Similarly, the gradient of the loss function with respect to b is given by

$$\frac{\partial Loss}{\partial b} = \frac{1}{11}\sum_{i=0}^{11}\left[2 \cdot (f(x_i) - y_i)\right].$$

Putting these together, the gradient vector is

$$\nabla Loss = \left[\frac{\partial Loss}{\partial a}, \frac{\partial Loss}{\partial b}\right] \approx [-76.07, -11.47].$$

Of course, if we simply change the a and b values by the negated gradient it is easy to see that our model would become a relatively steep line, and still not model the data properly. To account for this, a step size is introduced, we will initialize this to $\alpha = 0.01$. Then, the change our variables will undergo is

$$[a, b] \to \left[a - \alpha \cdot \frac{\partial Loss}{\partial a}, b - \alpha \cdot \frac{\partial Loss}{\partial b}\right]$$
$$[0, 0] \to [0.761, 0.115]$$

Simply doing this single iteration has now reduced the loss to a value of 3.71*. After a second iteration, we obtain a loss of* 0.60 *with parameters $a = 0.977$, and $b = 0.151$. After* 500 *iterations we obtain a loss of* 0.098 *with parameters $a = 0.943$, and $b = 1.004$. We can verify visually that this line models the data well in Fig. 5.5.*

Note that in Ex. 5.5 we had 11 pieces of data, and the loss function averaged the loss from each of them. For neural networks, it is common to either do this, or sum the loss attained from each individual piece of data.

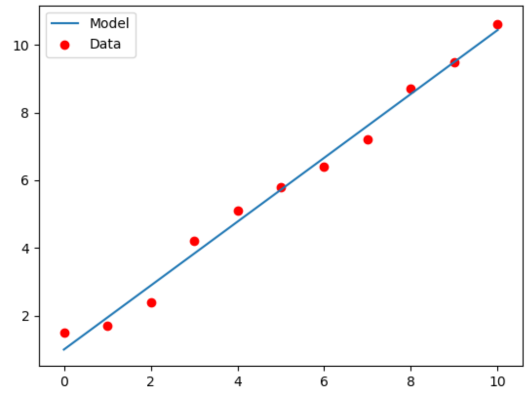


Figure 5.5: Results of using gradient descent for linear regression in Ex. 5.5.

The only differences between the above example and what happens with neural networks, is that neural networks are much more complicated functions with more parameters, and possibly multiple outputs, and neural networks typically use stochastic gradient descent. Dealing with the former first, we will begin by defining notation. Let $f(\boldsymbol{x};\boldsymbol{\theta})$ be a neural network with n layers where the i^{th} function has the form $\phi_i(\boldsymbol{w}^{(i)}\boldsymbol{x}^{(\boldsymbol{i})}+\boldsymbol{b}^{(i)})$, with ϕ_i being the activation function, $\boldsymbol{w}^{(i)}$ the matrix of weights, and $\boldsymbol{b}^{(i)}$ the vector of biases. Additionally, let z be the loss calculated from some differentiable loss function. With this notation, $\boldsymbol{w}^{(i)}_{j,k}$ is the weight of the connection of the k^{th} node in the $(i-1)^{th}$ layer, to the j^{th} node in the i^{th} layer. This can be seen by referring back to Ex. 5.2 in the activation functions section. There, when looking at the $\boldsymbol{w}_2$ matrix, we see that when we multiply it with $\begin{bmatrix} x & y \end{bmatrix}^\mathsf{T}$ on the right, the first column multiplies with the x and the second with the y. Hence, the column number of a weight in a weight matrix represents the node from the previous layer. Similarly, the row represents which node in the current layer the weight forms a connection with. We will also denote the value of the j^{th} node in the i^{th} layer with $\boldsymbol{x}^{(i)}_j$. Expanding the matrix-vector multiplication in each layer, we get the following formula for the value in each node:

$$\boldsymbol{x}_j^{(i)} = \phi_i\left(\sum_k \boldsymbol{w}_{j,k}^{(i)}\boldsymbol{x}_k^{(i-1)} + \boldsymbol{b}_j^{(i)}\right),$$

where k is the number of nodes in the $(i-1)^{th}$ layer. With this notation we can use the chain rule to calculate the gradient of the loss function with respect to any parameter in the network that we would like.

Example 5.6. *For this example we will work with a fully connected, n layer neural network that has two outputs. Suppose the second last layer of this network has four nodes, and we would like to calculate the gradient of the loss function, z, with respect to the weight connecting the fourth node in that second last layer, to the second node in the layer preceding it. Then, as can be visualized in Fig. 5.6, we must add the components of the derivative that we get from traversing every path connecting the loss function to that weight. In the given image, the weight that the partial derivative is being taken with respect to is shown in blue. The paths from the loss function to this weight are shown in red. This gives us the following partial derivative:*

$$\frac{\partial z}{\partial \boldsymbol{w}_{4,2}^{(n-1)}} = \frac{\partial z}{\partial \boldsymbol{x}_1^{(n)}}\frac{\partial \boldsymbol{x}_1^{(n)}}{\partial \boldsymbol{x}_4^{(n-1)}}\frac{\partial \boldsymbol{x}_4^{(n-1)}}{\partial \boldsymbol{w}_{4,2}^{(n-1)}} + \frac{\partial z}{\partial \boldsymbol{x}_2^{(n)}}\frac{\partial \boldsymbol{x}_2^{(n)}}{\partial \boldsymbol{x}_4^{(n-1)}}\frac{\partial \boldsymbol{x}_4^{(n-1)}}{\partial \boldsymbol{w}_{4,2}^{(n-1)}}$$

The first term is given from the upper two red lines, whereas the second term is given by the lower two red lines.

The process of updating the parameters of the network using the calculated gradients is called **backpropagation**. Data is fed forward through the network, the loss is calculated, and then the gradients are calculated and propagated backwards to update the parameters of the network; this is known as the learning process. It is in the backpropagation step of the learning process that we see the problem with the sigmoid function shown in the loss functions section.

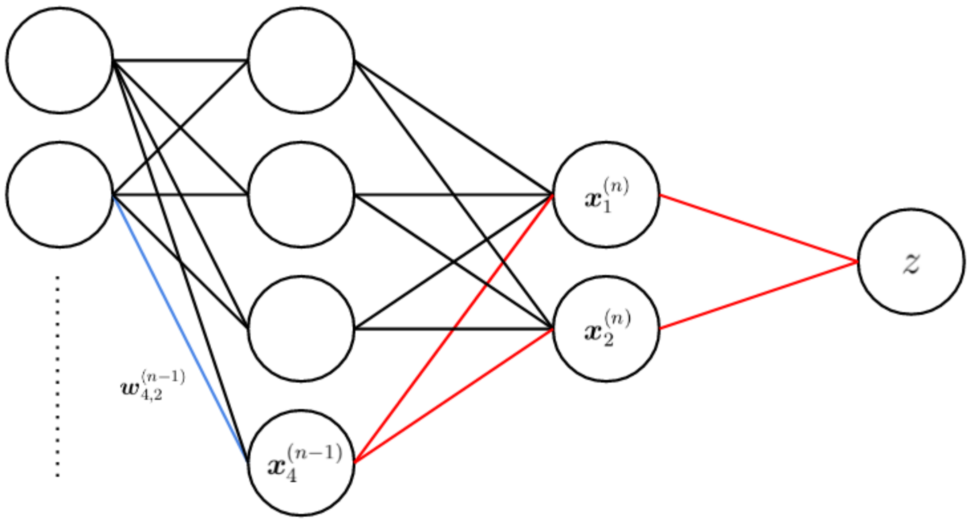


Figure 5.6: A visualization of how the gradient is calculated in Ex. 5.6. The weight that the gradient is being taken with respect to is shown in yellow. The paths connecting that weight and the loss function are shown in red.

Suppose we have a deep neural network where the same activation function is applied to each layer. Then, as we can see from the above example, each factor of each term, other than the partial derivatives of z with respect to a node, will be proportional to the derivative of the activation function. That is,

$$\frac{\partial \boldsymbol{x}_j^{(k)}}{\partial \boldsymbol{x}_i^{(k-1)}} \propto \phi'_k, \qquad \frac{\partial \boldsymbol{x}_i^{(k)}}{\partial \boldsymbol{w}_{i,j}^{(k)}} \propto \phi'_k.$$

It can be seen that if we had taken the gradient with respect to a weight one layer nearer the input, there would be an additional factor on each term as well. Thus, the gradient of the loss function with respect to any given weight is proportional to the derivative of the loss function to the power of the number of layers between the weight and the end of the network;

$$\frac{\partial z}{\partial \boldsymbol{w}_{i,j}^{(k)}} \propto (\phi')^{(n-k)}.$$

As we can see from Fig. 5.3, if we repeatedly multiply the derivative of the sigmoid function with itself, it will approach the $y = 0$ line. Hence, the gradient with respect to parameters near the input will be negligible no matter the loss, and it will not be possible to update them efficiently. Thus, the sigmoid function, and related activation functions such as the hyperbolic tangent function, would not be good choices for deep neural networks. This issue is known as the **vanishing gradient problem**.

The back propagation step is also where the issue of dead nodes with the ReLU activation function arises. Specifically, the partial derivative of a node with respect to a weight will result in an expression of the form

$$\frac{\partial \boldsymbol{x}_i^{(k)}}{\partial \boldsymbol{w}_{i,j}^{(k)}} = \frac{\partial \phi_k}{\partial \boldsymbol{w}_{i,j}^{(k)}} \cdot \boldsymbol{x}_j^{(k-1)},$$

where if ReLU is the activation function, and its input happens to be non-positive, then

$$\frac{\partial \phi_k}{\partial \boldsymbol{w}_{i,j}^{(k)}} = 0.$$

Thus, any weight in an affine transformation resulting in a negative value will not be updated if the ReLU activation function is used. In addition, this is why the variants of the ReLU function attempt to overcome this problem by maintining a nonzero derivative for negative inputs.

Given a good choice of activation function, this method for updating the weights should work well in situations where we can run all the data through the model at once. However, we typically do not have all possible data, and there is often too much to run through at once. For example, we cannot possibly attain every person's unique way of writing letters and numbers, but there are databases available that contain many thousands of such examples. It is infeasible, though, to give a model say 100,000 images as they, as well as the model with all of its parameters, would take up more memory than the graphics card can hold (most machine learning training is performed on a computer's graphics card). For this reason, neural networks typically perform stochastic gradient descent (SGD) throughout the learning process.

Definition 5.12 (Stochastic Gradient Descent)**.** *Stochastic gradient descent uses gradient descent to minimize a function, but only gives the model a randomly selected portion of the data at a given step of the algorithm.*

Example 5.7. *In this example we will perform the same task, using the same data, as in Ex. 5.5. Instead of using regular gradient descent, though, we will make use of SGD. This will be done by only giving the model two data points at a time. For the first iteration, we will give the model* $(0,\ 1.5)$ *and* $(1,\ 1.7)$*. This results in a loss of*

$$Loss = \frac{1}{2}\left[(f(0) - 1.5)^2 + (f(1) - 1.7)^2\right] = 2.57$$

Calculating the gradient in the same way, and with the same step size of $\alpha = 0.01$*, the first parameter update results in* $a = 0.017$*, and* $b = 0.032$*. Next, giving the model the points* $(2,\ 2.4)$ *and* $(3,\ 4.2)$*, we get a loss of* 11.20*, producing the parameters* $a = 0.187$*, and* $b = 0.097$*. Continuing to give the model the data in a sequential order, after* 500 *iterations we obtain the values* $a = 0.913$*, and* $b = 0.934$ *with a loss of* 0.168*, giving the results shown in Fig. 5.7.*

As we can see from this new example, SGD performs quite differently than regular gradient descent. To begin with, if it were possible to have an infinitesimal step size, the loss for regular gradient descent would be monotonic decreasing. However, this is not the case for SGD. This is because each batch of data the model receives does not necessarily reflect the entire dataset. Thus, SGD has a lower convergence rate than regular gradient descent [15]. In the examples, after 500 iterations, the fit of the model for SGD is not as good as that of

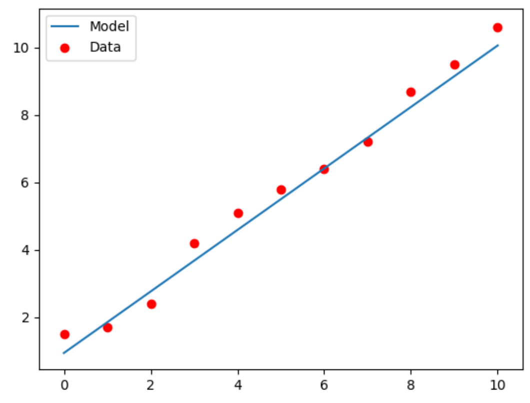


Figure 5.7: Results of using stochastic gradient descent for linear regression in Ex. 5.7.

regular gradient descent. This can be quantified numerically by calculating the loss of the SGD model with the full dataset, giving 0.158. There are several methods we can apply to address these problems, and improve the performance of SGD.

The first thing we can do to improve the SGD algorithm is to randomly pick, or shuffle the data points that we give the model, as opposed to providing them sequentially. Doing so results in the model having to minimize the loss produced from many different batches, which provides a wider variety of gradient directions. Globally shuffling the dataset after enough steps of SGD to get through all of it, and sampling the batches without replacement has been shown to provide a speedup in the training time [15]. Implementing this tactic on our SGD model, we reduce the loss on the full dataset from 0.158 to 0.114. This new loss value was averaged over 1,000 trials due to the introduction of the random sequence of data points possibly affecting how well the parameters are updated.

Definition 5.13 (Hyperparameter). *A hyperparameter is a variable that is set ahead of the learning process; gradient descent does not affect these parameters.*

Another simple way one can optimize the SGD algorithm is by fine-tuning the hyperparameters. As of now, the regression examples have several hyperparameters; the number of iterations, the batch size, and the step size. In terms of neural networks, the width and depth of the model are included as well. These parameters can drastically affect how quickly a model converges, if at all. For example, if we double the number of iterations, this now improves the average total loss for the SGD algorithm from our example from 0.114 to 0.099, whereas if we half it, we obtain an average loss of 0.134. This now almost matches the results from regular gradient descent! However, we can further improve the model as the hyperparameters used for the regular gradient descent example were not fine-tuned as well.

The step size, or more often called **learning rate** in the realm of neural networks, is another very influential hyperparameter. It can be difficult to find the appropriate value for the learning rate as if it is too large, the algorithm may either oscillate around the solution, but not produce an accurate answer, or diverge entirely. Conversely, if it is too small, the algorithm may never converge, get stuck in a local minimum, or take much longer than necessary to converge. These can be seen in the regular gradient descent method shown in Ex. 5.5 where even if the step size were $\alpha = 0.05$, by the tenth iteration, the model already has a loss on the order of 10^8 with parameter values of $a \approx 13{,}501$, and $b \approx -1{,}944$. However, if the learning rate were $\alpha = 0.001$, it would take approximately 4,000 iterations to achieve the same loss shown in the example. This is not to say that $\alpha = 0.01$ is optimal, though. The best learning rate for this example is in the neighborhood of $\alpha = 0.025$, which achieves the same loss as in the example in approximately 200 iterations. While for this trivial example it may not make a large difference, for complicated models it can be the difference between training a model for hours, compared to days.

Methods of finding the optimal hyperparameters is an active area of research. Typically, one discriminates between sets of hyperparameters by training one's model for several iterations through the entire dataset, and testing its accuracy. Accuracy is calculated differently depending on the purpose of the network. For classification models, accuracy is given by the frequency with which a model correctly classifies data on a particular dataset. For models that predict values, one can use tolerance levels; the accuracy is given by the frequency that a model correctly produces values in a given tolerance on a particular dataset. The dataset that is used to measure accuracy is further discussed in section 5.6. A popular method for generating sets of hyperparameters is through a grid search [8].

Definition 5.14 (Grid Search)**.** *To perform a grid search, one generates subsets of each parameter that they would like to test. Then, each element of each hyperparameter subset is tested with all variants of the other hyperparameters.*

As of now, we have only covered good practices to optimize the SGD algorithm, however improvements on the algorithm itself exist as well. We present several optimization techniques, all of which can be found in [8].

Stochastic Gradient Descent With Momentum

One such method of improving the SGD algorithm is to incorporate the "momentum" of the gradient in the parameter change. This is done by storing previous gradient values and calculating a moving average by exponentially weighing them. Let z_k be the loss calculated on the k^{th} iteration of training. Then instead of updating the parameters by the learning rate multiplied with the gradient of z_k with respect to its parameters, we update it with

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \alpha \boldsymbol{V}_k,$$

where α is the learning rate, $\boldsymbol{\theta}_k$ are the parameters of the network on the k^{th} iteration of training, and $\boldsymbol{V}_k$ is given by

$$\boldsymbol{V}_k = \beta \boldsymbol{V}_{k-1} + (1-\beta)\nabla z_k,$$

where $\boldsymbol{V}_0$ is the zero vector, $\beta \in (0, 1)$ is called the momentum coefficient, and ∇z_k is the gradient of the loss function with respect to the network's parameters. Note that β is another hyperparameter of the model, effectively defining the weight of previous gradients relative to the current one. The motivation behind adding this momentum term is to be able to get out of local minima, just like rolling a ball down a hill with enough momentum can overcome rocks that get in its way. This method of updating parameters of the model is called SGD with momentum (SGDM).

AdaGrad

Another common method for improving SGD is an algorithm called AdaGrad. Changing certain hyperparameters of the model during the learning process is common practice and offers many advantages. The AdaGrad algorithm changes the learning rate based on the size of the gradient. One advantage of doing this, is that it decreases the model's sensitivity to the initial learning rate. That is, if you were to pick a sub-optimal learning rate, the model would likely not perform as bad as if the learning rate were not to be adjusted. Though, if an optimization algorithm such as this is not implemented, it is common practice to manually adjust the learning rate when one sees fit. The way AdaGrad specifically scales the learning rate, is specific to each parameter. That is, for a given parameter of the network, the learning rate is divided by the Euclidean norm of all previous gradient values with respect to that parameter. Since the gradient with respect to each parameter is different, each parameter then has a different learning rate. When the gradient with respect to a given parameter is large, that parameter receives a greatly decreased learning rate. Similarly, the parameters corresponding to the smaller components of the gradient receive relatively small decreases in their learning rates. This algorithm is theoretically desirable for convex optimization as there is then greater progress for those parameters with smaller slopes. The problem with this algorithm is that the accumulation of all squared gradients from the beginning of training can become excessively large for deep neural networks, effectively preventing the model from learning, after a given number of iterations.

RMSProp

Combining ideas from both SGDM and Adagrad we can obtain an algorithm called RMSProp. RMSProp takes the concept of exponentially weighted gradients from SGDM and applies it to the learning rate modification of AdaGrad. In other words, for a particular parameter of the network, one divides the learning rate by the Euclidean norm of all previous gradient values with respect to that parameter, but exponentially weighs each gradient value in favour of those that are more recent. This is similar to allowing the network only a finite memory of the previous gradients in the AdaGrad algorithm, fixing the problem of overly decreased learning rates. This extension of AdaGrad then allows it to work well in nonconvex optimization as well. RMSProp has been shown experimentally to perform well in many situations, and is one of the default optimization methods that can be applied to almost any problem.

Adam

The incorporation of gradient momentum for the parameter updates themselves, along with RMSProp leads to an algorithm called Adam. Named so after adaptive moments, Adam also includes a correction to the updates on the gradients of the weights corresponding to biases. Note that since Adam has both momentum incorporated in the gradient as well as the scaling factor for the learning rate, it possesses one momentum coefficient for each of them. Empirically, Adam has been found to work well for many different models. Additionally, this algorithm has proven quite robust with respect to hyperparameters, allowing them to be kept at their defaults in most cases, apart from the learning rate. For more on Adam, see [8, 16].

5.5 Convolutional Neural Networks

One common application of neural network models is image recognition and classification. The goal of such applications is as follows: Given an image and a potential set of possible labels, assign the label(s) to that image that agrees with what human eyes would see represented in that image. Some models go so far as to surround objects in the image with a box that labels it. However, images include much more information, and therefore, more complications. Suppose we want to create a fully connected neural network model that takes in a 200×200 image, has one hidden layer of the same width as the input layer, and then leads to a single node output. This model would have 4×10^4 input nodes, one for each pixel value; 4×10^4 hidden layer nodes; and one output node. That means we need the computer to store $4 \times 10^4 \times 4 \times 10^4 + 4 \times 10^4 \approx 1.6 \times 10^9$ weight values! If each of these weight values are stored as the usual 8-byte, double precision floating point numbers, this model would take up almost 12GB of data. Keep in mind that this is for a relatively small image, and does not even include multiple hidden layers, bias nodes, or other values the model needs to store. The way we solve this computer memory problem is through convolutional neural networks (CNNs).

As the name suggests, CNNs make use of a convolution operation to reduce the number of parameters in the network. Specifically, CNNs have a **filter**, or **kernel** possessing weights like those from fully connected networks that get updated through gradient descent. This filter passes over the image and computes a discrete convolution. The output from the discrete convolution is then often called a feature map, for reasons we will soon see. If we take I to be our input image, and K to be the two dimensional filter, then the discrete

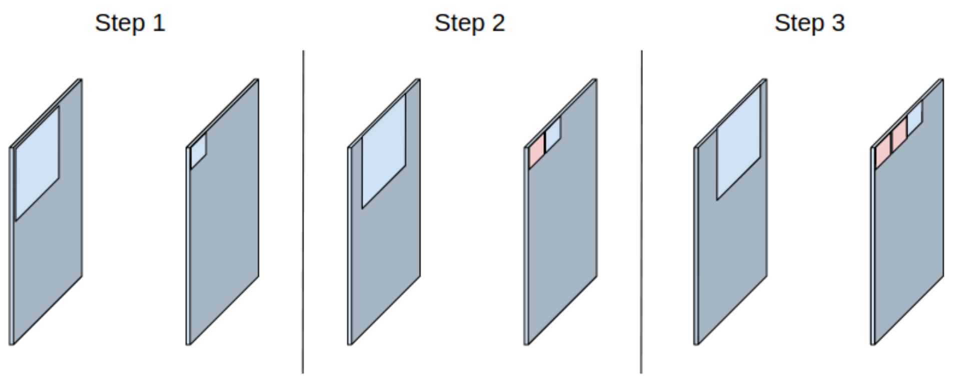


Figure 5.8: A visual aid to help understand what exactly the cross correlation operation in CNNs computes. In each step, the light blue square in the left image is the kernel, whereas the light blue square in the right image is the output of the cross correlation. The image on the right of each step is then the feature map.

convolution is given by

$$(I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m,j-n) = \sum_m \sum_n I(i-m,j-n)K(m,n),$$

where m and n run through the dimensions of the kernel, and i and j are selected so that the filter is always on the image. In reality though, this commutivity property is because the kernel is flipped relative to the input, and so most neural network implementations actually compute the cross-correlation [8]

$$(K * I)(i,j) = \sum_m \sum_n I(i+m,j+n)K(m,n).$$

This operation can be visualized in Fig. 5.8. In each step, the light blue square in the left image represents the filter passing over the image. Every pixel value inside this filter will get multiplied by a weight, and summed to create the single value stored in the light blue square on the right image. The filter then moves over and computes the next cross-correlation. Each step computes a single value in the feature map; the image on the right.

Example 5.8. *Let the input matrix I and the kernel matrix K be defined as*

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \qquad K = \begin{bmatrix} 0.4 & 0.3 \\ 0.2 & 0.1 \end{bmatrix}.$$

The cross-correlation $(K * I)(1,1)$ can then be computed as

$$(K * I)(1,1) = \begin{bmatrix} 1 \times 0.4 & + & 2 \times 0.3 \\ 5 \times 0.2 & + & 6 \times 0.1 \end{bmatrix} = 2.6.$$

This is only one entry of the feature map, though. The filter then passes over the whole image computing cross-correlation at each step. The full feature map for this example is then

$$\begin{bmatrix} (K*I)(1,1) & (K*I)(1,2) & (K*I)(1,3) \\ (K*I)(2,1) & (K*I)(2,2) & (K*I)(2,3) \\ (K*I)(3,1) & (K*I)(3,2) & (K*I)(3,3) \end{bmatrix} = \begin{bmatrix} 2.6 & 3.6 & 4.6 \\ 6.6 & 7.6 & 8.6 \\ 10.6 & 11.6 & 12.6 \end{bmatrix}$$

We call the output of the convolution with the kernel a feature map because it can be thought of as just that. The kernel essentially computes how strong the presence of a given feature is at the location on the image where the cross correlation is computed. For example, the kernel

$$\begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$$

can be thought of as a vertical line detection filter. Then, as it passes over all locations of the image, it computes a value corresponding to the likelihood of there being a vertical line in that section of the image. The entire output then maps out where the filter detected lines, and where it did not.

The kernels do not begin with predetermined filters but instead learn to detect important features to the dataset the network is trained on. Near the start of the network, we get filters which may detect lines, or other obscure properties. As we move deeper through the network, kernels then act on feature maps, allowing them to detect more complex objects. For example, if a vertical line was detected beside a horizontal line, we may have a corner, or something similar in the image. The build up of these complex features from more simple ones is what allows CNNs to read hand-written digits, distinguish between dogs and cats, and perform well in many other useful applications [8].

With these convolution operations we can still make use of many of the same tactics as before, such as including biases and applying non-linear activation functions. The only difference is that we save a significant amount of memory by reducing the parameters in the network. In fact, we save so much memory that most networks typically have many different filters in the same layer, each one producing a different feature map, as each one picks up on different features. However, this can often lead to confusion with how each feature map in the next layer is actually calculated. For a kernel acting on a layer with multiple feature maps, each feature map has a different weight in the kernel. So, the cross correlation computation is now given by

$$(K * I)(i, j) = \sum_{d} \sum_{m} \sum_{n} I(i + m, j + n, d) K(m, n, d).$$

where d goes over all feature maps. In other words, if we think of the feature maps and filter as three dimensional arrays, we multiply them component-wise for a particular i, and j, and sum all the values. To obtain a layer with multiple feature maps, one then uses multiple three dimensional filters.

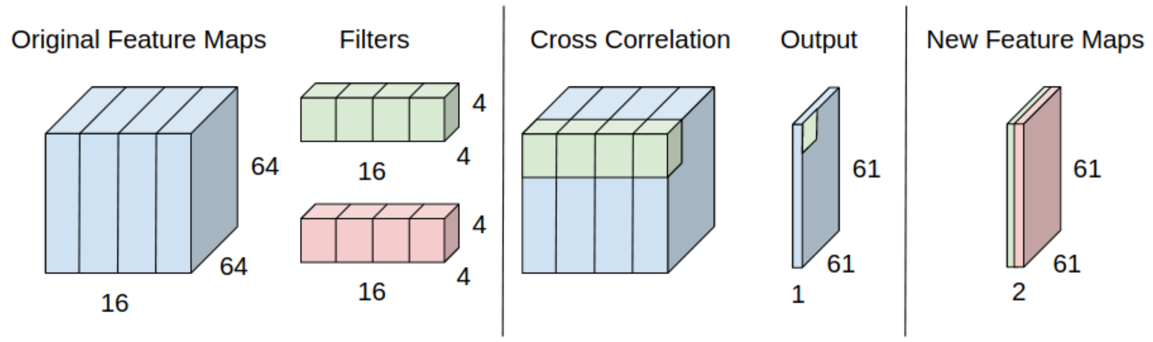


Figure 5.9: A visual depiction of a filter that acts on multiple feature maps

Example 5.9. *Suppose that one layer outputs an object of size* $16 \times 64 \times 64$*, that is,* 16 *feature maps of height and width* 64*. If the next layer has two* 4×4 *filters, each one actually* $16 \times 4 \times 4$ *weights, we simply do not mention the 16 as it is always made to match the number of feature maps in the previous layer. Each feature map of the previous layer then receives its own* 4×4 *convolution. After the single-value producing cross correlation over the three dimensional array, the filter goes through all locations on the original feature maps. An example of this can be seen in Fig. 5.9.*

We can further reduce the number of parameters in our network by incorporating components from the following definitions in our network.

Definition 5.15 (Stride)**.** *The stride is a property of a filter that defines how far it moves before computing the next cross-correlation. A filter with a stride of two would compute a cross-correlation, move over two pixels, and repeat.*

Definition 5.16 (Pooling Layer)**.** *Pooling layers are layers in the network with the purpose of reducing the height and width of their input. This is done by increasing the stride of the filter.*

Definition 5.17 (Max Pooling)**.** *A max pooling layer is a type of pooling layer that returns the largest value inside the filter.*

Definition 5.18 (Average Pooling)**.** *An average pooling layer is a type of pooling layer that returns the average value inside the filter.*

Note that max pooling and average pooling layers act on all feature maps, but do so one at a time, preserving the number of feature maps from the input. Additionally, these layers are not included in the depth of the neural network.

The cross-correlation shown in Ex. 5.8 can be considered an example of pooling as it reduced the size of the input from a 4×4 matrix to a 3×3 matrix. However, most pooling layers achieve the reduction in size through a stride greater than one, and typically cut the height and width of the input in half, approximately. Another reason pooling can be extremely useful in image related applications, is that it may make the network invariant to small translations [8]. This allows networks to be able to detect what an object is no matter where it is placed on a given image.

Example 5.10. *Using the same input matrix and filter as in Ex. 5.8, we get the following outputs when using a stride of two both horizontally and vertically.*

$$\begin{array}{ccc} \textit{Original Filter} & \textit{Max Pooling} & \textit{Average Pooling} \\ \begin{bmatrix} 2.6 & 4.6 \\ 10.6 & 12.6 \end{bmatrix} & \begin{bmatrix} 6 & 8 \\ 14 & 16 \end{bmatrix} & \begin{bmatrix} 3.5 & 5.5 \\ 11.5 & 13.5 \end{bmatrix} \end{array}$$

Note that in the above example, all of the heights and widths of the feature maps were cut in half. Regular filters with a stride larger than one can be a useful method of pooling as it allows the network to learn what is important when reducing the size of the feature map. Additionally, notice that if we had chosen a stride of three, we would have difficulty computing the output due to the filter being only half on the image. Strides and filter sizes are chosen to avoid this dilemma.

With the current framework, we can reduce the height and width of an image quite easily, in fact, it is the only thing we can do. So, how can we maintain the dimensions of an image? This is done through zero padding. Essentially, we add a border of zeros around the outside of the image. We can add more than one border of zeroes as well. Note that a border of width one actually increases the width and height of a layer by two, as there will be zeroes on both sides of the data. This not only allows us to maintain the same dimensions, but allows the edges of the image to be involved in calculations with the same frequency as pixels in the middle of the image. If we want to maintain the size of the input with a filter of width f, we need a border of width $(f-1)/2$. Though, we require an integer from this calculation, hence, only filters with odd valued widths and heights can maintain image dimensions. If a given layer in our CNN has a width or height given by m, then the corresponding dimension of the feature map, m', will be

$$m' = \frac{m - f + 2p}{s} + 1,$$

where p is the zero padding border width, s is the stride of the filter, and f is the corresponding length of the filter. We require that this always results in an integer so that the filters only ever fully fit on the input and border.

5.6 Applications To Hand-Written Digit Recognition

Now that we have all of the necessary tools to construct and train a neural network, we will go through the process of creating one, and train it to recognize hand-written digits. The data from this section comes from the MNIST library, which contains 60,000 images for training, and 10,000 images for testing, all of size $1 \times 28 \times 28$. The "1" in the size represents the colour channel, meaning that it is purely black and white. It is common practice to split up a dataset into both a **training set** and **testing set**: The model is exclusively trained on the training set, and once it is done training, it is measured for accuracy on the testing set. It is important to note that backpropogation is not performed when testing a model for accuracy, and the testing set is only used once, after training is complete. However, with this setup, we cannot test various sets of hyperparameters through a grid search, or any other search method: We cannot use the testing set as this was used to update the parameters, and we cannot use the testing set as the accuracy would then no longer reflect how well the model generalizes. The testing set is likely to contain biases towards some features of the true distribution, hence fine-tuning the hyperparameters to it would likely fit to those biases as well. One solution to this problem is to include a third set called the **validation set**. This set is used to measure accuracy during training. A consistently decreasing accuracy on the validation set is an indication that the model is beginning to overfit to the training set. With this validation set, the accuracy on the testing set will still provide an unbiased estimate of the generalization accuracy of the model.

For this model we will define a convolutional layer to consist of: A dimensionality maintaining 5×5 kernel with two layers of zero padding, or a border of width two, followed by the ReLU activation function, and a 2×2 max pooling layer with a stride of two that cuts the height and width in half. Then the architecture of the network we are going to use to recognize hand-written digits is as follows, and can be visualized in Fig 5.10:

- Input the $1 \times 28 \times 28$ image
- Apply a convolutional layer resulting in 16 feature maps of size 14×14
- Apply a convolutional layer resulting in 32 feature maps of size 7×7
- Flatten the $32 \times 7 \times 7$ feature maps to a 1,568 node vector
- Feed the data forward through a fully connected layer to a 100 node vector and apply the ReLU function

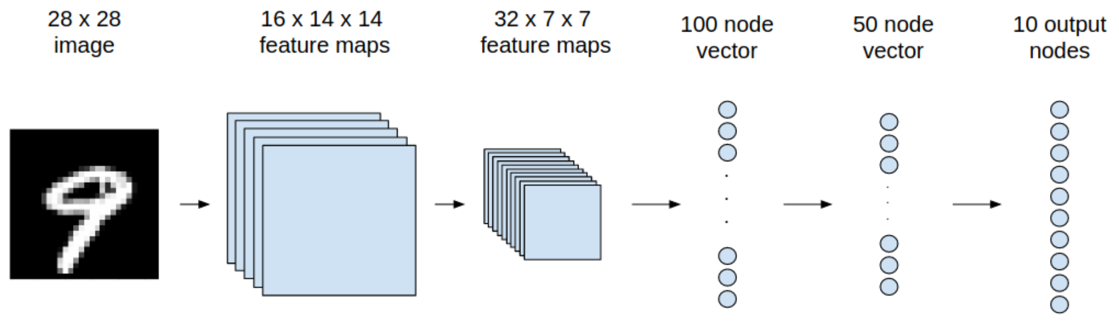


Figure 5.10: A visual depiction of the model used to classify hand written digits.

- Feed the data forward through a fully connected layer to a 50 node vector and apply the ReLU function
- Feed the data forward through a fully connected layer to a 10 node output vector

Each of the values in the output vector is then associated with a number; the first element pertains to zero, the second to one, and so on. Before training the network, the image of the nine shown in Fig 5.10 gives the output

$$\begin{bmatrix} 0.142 & -0.045 & 0.043 & 0.076 & 0.030 & -0.068 & -0.090 & 0.134 & 0.019 & -0.081 \end{bmatrix}^{\mathsf{T}}.$$

Applying the softmax function would suggest that the number is a one with a probability of 11.3%. Using the CE loss with this output, we get an error of

$$CE(-0.0810) = -\ln\left(\frac{e^{-0.0810}}{\sum_i e^{x_i}}\right) = -\ln\left(\frac{0.922}{10.195}\right) = 2.403$$

To train the model, we made use of the Adam optimizer and the CE loss. We trained the model for 20 **epochs**; the number of times run through the entire training set, and used the following hyperparameters:

- A learning rate of 0.0001
- A momentum coefficient of 0.9 in the gradient
- A momentum coefficient of 0.999 in the learning rate scaling factor
- A batch size of 100 images

After performing these calculations, the new output for the image of a nine shown above is

$$\begin{bmatrix} -15.25 & -18.01 & -11.52 & -3.83 & 4.09 & -10.98 & -34.18 & 4.68 & -0.60 & 17.88 \end{bmatrix}^{\mathsf{T}}$$

Applying the softmax function, this corresponds to the model predicting that the image is a nine with a probability of 99.99%. The true accuracy of our model, though, is given from how well it performs on the test set. For this configuration, the model obtained an accuracy of 98.92%.

5.7 Regularization Methods

As with many models, neural networks are prone to overfitting data. This problem is made worse as we do not know the complexity necessary to accurately fit the data. Further, many of the problems neural networks are applied to require a model complexity similar to simulating the entire universe [8]. Because of this, the best deep learning models are ones that are large, but apply methods that prevent it from overfitting to the training data [8]. These are known as regularization methods.

One simple methodology we can put into practice to prevent overfitting is to obtain as much data as possible, however, this is not always realistic or easy. As mentioned previously, we can also implement the use of a validation set, and stop training if the accuracy becomes consistently decreased. Throughout this section we present algorithm modifications that can be applied in order to help a given model generalize to data that it was not trained on. Note that this often comes at the expense of a higher loss on the training set [8].

Parameter Norm Penalties

One common method of regularization is adding a parameter norm penalty. That is, we add a term to the loss function based on a given norm of the parameters:

$$\tilde{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = L(\boldsymbol{x}, \boldsymbol{y}) + \gamma\Omega(\boldsymbol{\theta}),$$

where L is the loss function with the tilde indicating the use of regularization, $\boldsymbol{x}$ is the true output, and $\boldsymbol{y}$ is the model output. Here, Ω calculates a norm of the parameters $\boldsymbol{\theta}$, and $\gamma \in [0, \infty)$, another hyperparameter, controls the extent of its effect on the loss. Note that when implemented, we typically only add a norm penalty to the weights of the network and not the biases. This is because regularizing the biases as well typically leads to underfitting the data [8]. Hence, we can replace $\boldsymbol{\theta}$ with $\boldsymbol{w}$ in this equation. This may not seem like an intuitive thing to do, why would we want to limit the weights values that our model can take on? However, larger weight values are typically a sign that the network is beginning to overfit [8].

Goodfellow et al. [8] used two common norms for this type of regularization; the L1 norm, and half the square of the L2 norm, also known as Tikhonov regularization, ridge regression, and weight decay. These modify the gradient as,

L1 Regularization	L2 Regularization
$\tilde{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = L(\boldsymbol{x}, \boldsymbol{y}) + \gamma\lvert\lvert\boldsymbol{w}\rvert\rvert_1$	$\tilde{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = L(\boldsymbol{x}, \boldsymbol{y}) + \frac{\gamma}{2}\boldsymbol{w}^\mathsf{T}\boldsymbol{w}$
$\nabla\tilde{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = \nabla L(\boldsymbol{x}, \boldsymbol{y}) + \gamma\,\mathrm{sign}(\boldsymbol{w})$	$\nabla\tilde{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = \nabla L(\boldsymbol{x}, \boldsymbol{y}) + \gamma\boldsymbol{w}$

where $||\boldsymbol{w}||_1$ is the sum of the absolute value of each weight parameter. The update to the weight parameters of the network is then,

$$\boldsymbol{w} \rightarrow \boldsymbol{w} - \alpha\gamma \operatorname{sign}(\boldsymbol{w}) - \alpha\nabla L(\boldsymbol{x}, \boldsymbol{y}) \qquad\qquad \boldsymbol{w} \rightarrow (1 - \alpha\gamma)\boldsymbol{w} - \alpha\nabla L(\boldsymbol{x}, \boldsymbol{y}),$$

which can still be subject to optimization algorithms. To see how these methods affect the network over the entire course of training, we can approximate the loss function with a second order Taylor series with respect to $\boldsymbol{w}$ about the minimum of L, $\boldsymbol{w}^*$, giving

$$\hat{L}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^{\mathsf{T}} \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}^*)(\boldsymbol{w} - \boldsymbol{w}^*),$$

where $\boldsymbol{H}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}^*)$ is the Hessian matrix of L with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. Note that the first order term is not present in this equation as it vanishes at $\boldsymbol{w}^*$. In addition, if the loss function truly were quadratic in the neighborhood of of the minimum of L, such as in regression by mean squared error minimization, then this would not be an approximation at all [8]. For simplicity, we will now omit the $\boldsymbol{x}$ and $\boldsymbol{y}$ arguments. The gradient of our approximated loss function with respect to the weights of the network is now

$$\nabla \hat{L}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*).$$

If we now add the L2 regularization term and attempt to solve for the new minimum at $\boldsymbol{w}'$, we get that

$$\begin{gathered}\gamma \boldsymbol{w}' + \boldsymbol{H}(\boldsymbol{w}' - \boldsymbol{w}^*) = 0 \\ \boldsymbol{w}' = (\boldsymbol{H} + \gamma \boldsymbol{I})^{-1} \boldsymbol{H} \boldsymbol{w}^*.\end{gathered}$$

Note that $\boldsymbol{H}$ was evaluated at $\boldsymbol{w}^*$, the minimum of L, and hence it is positive semi-definite and symmetric. Thus, we can perform an eigenvalue decomposition of the form $\boldsymbol{H} = \boldsymbol{Q\Lambda Q}^\intercal$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues, and $\boldsymbol{Q}$ is an orthogonal matrix of eigenvectors. Hence our new solution $\boldsymbol{w}'$ becomes

$$\begin{aligned}
\boldsymbol{w}' &= (\boldsymbol{Q\Lambda Q}^\intercal + \gamma\boldsymbol{I})^{-1}\boldsymbol{Q\Lambda Q}^\intercal\boldsymbol{w}^* \\
&= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \gamma\boldsymbol{I})\boldsymbol{Q}^\intercal\right]^{-1}\boldsymbol{Q\Lambda Q}^\intercal\boldsymbol{w}^* \\
&= \boldsymbol{Q}^{-\top}(\boldsymbol{\Lambda} + \gamma\boldsymbol{I})^{-1}\boldsymbol{Q}^{-1}\boldsymbol{Q\Lambda Q}^\intercal\boldsymbol{w}^* \\
&= \boldsymbol{Q}(\boldsymbol{\Lambda} + \gamma\boldsymbol{I})^{-1}\boldsymbol{\Lambda Q}^\intercal\boldsymbol{w}^*.
\end{aligned}$$

Thus, the effect of L2 regularization is to rescale the optimal solution of L, $\boldsymbol{w}^*$, with the eigenvectors of the Hessian matrix of the loss function. Specifically, if we write $\boldsymbol{w}^*$ and $\boldsymbol{w'}$ as $\boldsymbol{w}^* = \boldsymbol{Qp}$ and $\boldsymbol{w'} = \boldsymbol{Qq}$, then the above equation becomes

$$\boldsymbol{q} = (\boldsymbol{\Lambda} + \gamma\boldsymbol{I})^{-1}\boldsymbol{\Lambda p}.$$

Hence $\boldsymbol{p}_i$, which is the coordinate of $\boldsymbol{w}^*$ in the i^{th} eigenvector direction of $\boldsymbol{H}$, is rescaled as $\lambda_i/(\lambda_i + \gamma)$. Shifts in directions that do not contribute significantly to reducing the loss, which corresponds to a small eigenvalue ($\lambda \ll \gamma$) in the Hessian matrix, then decay away. Similarly, for steps in directions that do contribute significantly to lowering the loss, the eigenvalue ($\lambda \gg \gamma$) is large enough that the step in that direction is relatively unaffected. The logic behind this is that in order to have a significant effect on the loss, a given step is more likely to be beneficial to many data points. Hence, the steps that get decayed away are more likely to be those in directions only beneficial to a few data points, which results in overfitting.

This rescaling effect was only the cause of L2 regularization, though, so how does L1 regularization affect the network throughout the training process? First of all, L1 regularization shifts all parameters of $\boldsymbol{w}$ by the same amount, unlike L2. In order to accomplish the same level of analysis with these shifts, we need the approximation that $\boldsymbol{H}$ is diagonal and positive definite. This approximation holds if for a linear regression problem, one pre-processes that data in order to remove all correlation between input features. One can use principal component analysis to achieve this [8]. With this approximation we have

$$\hat{L}(\boldsymbol{w}) = L(\boldsymbol{w}^*) + \sum_i \left[\frac{1}{2}(\boldsymbol{w}_i - \boldsymbol{w}_i^*)^\mathsf{T} \boldsymbol{H}_{i,i}(\boldsymbol{w}_i - \boldsymbol{w}_i^*) + \gamma|\boldsymbol{w}_i|\right].$$

Following the same procedure as with L2 regularization, we want to optimize this approximation. We get the analytic solution of

$$\frac{\partial \hat{L}(\boldsymbol{w})}{\partial \boldsymbol{w}_i} = (\boldsymbol{w}_i - \boldsymbol{w}_i^*)\boldsymbol{H}_{i,i} + \gamma \text{sign}(\boldsymbol{w}_i) = 0.$$

Note that if $\boldsymbol{w}_i < 0$, then $\boldsymbol{w}_i^* < -\gamma/\boldsymbol{H}_{i,i}$, and if $\boldsymbol{w}_i > 0$, then $\boldsymbol{w}_i^* > \gamma/\boldsymbol{H}_{i,i}$. Hence, $\text{sign}(\boldsymbol{w}_i) = \text{sign}(\boldsymbol{w}_i^*)$. Rearranging for $\boldsymbol{w}_i$ we see

$$\boldsymbol{w}_i = \boldsymbol{w}_i^* - \text{sign}(\boldsymbol{w}_i^*)\frac{\gamma}{\boldsymbol{H}_{i,i}} = \text{sign}(\boldsymbol{w}_i^*)\left(|\boldsymbol{w}_i^*| - \frac{\gamma}{\boldsymbol{H}_{i,i}}\right).$$

However, there is no reason that the restriction $|\boldsymbol{w}_i^*| > \gamma/\boldsymbol{H}_{i,i}$ should exist. Therefore this must occur in the only case we have not considered; $\boldsymbol{w}_i = 0$. Hence the solution is

$$\boldsymbol{w}_i' = \text{sign}(\boldsymbol{w}_i^*) \max\left(|\boldsymbol{w}_i^*| - \frac{\gamma}{\boldsymbol{H}_{i,i}}, 0\right).$$

The case where $|\boldsymbol{w}_i^*| \leq \gamma/\boldsymbol{H}_{i,i}$ and hence $\boldsymbol{w}_i' = 0$ occurs, is when the portion of L in $\tilde{L}$ is overwhelmed by the regularization term, and is driven towards zero. We also see that if $|\boldsymbol{w}_i^*| > \gamma/\boldsymbol{H}_{i,i}$, then the solution to the regularized loss function will be shifted by $\gamma/\boldsymbol{H}_{i,i}$ towards zero from the solution to the non-regularized loss function. The overall effect of

L1 regularization is then an increase in sparsity of the weight parameters. This has been taken advantage of in order to try and simplify models [8]. For example, one could run a model through training with L1 regularization, observe which weights become zero, or close to it, and that indicates that those parameters of the network are likely to be unnecessary. Thus, we can safely remove those nodes, or feature maps from the network without greatly affecting the loss or efficiency of learning. One could also incorporate a combination of both L1 and L2 loss, however, we will not use or analyze this method.

Dropout

Another common regularization technique is called dropout. It is the process in which during training, we randomly drop nodes according to a probability. The dropout probability is yet another hyperparameter for the model. The difference between this and dropping the nodes based on their weight from L1 regularization, is that dropping these nodes is not permanent. Instead, different nodes are dropped for each batch of training, and the nodes chosen are not due to their weight value, but a dropout probability. Implementing dropout also brings about the idea of subnetworks; a network constructed from a subset of the nodes of the full model. Additionally, dropout increases the speed at which we can train our network as a subnetwork has less parameters than the full model, and therefore there are less values and gradients to compute. It is important to note that the output nodes have zero probability of being dropped, as this would likely produce poor results, increase the loss, and drive the parameters of the network further from their optimal values. Additionally, dropout is only applied during the learning process, not when using and testing the model.

Dropout works on the assumption that if all the subnetworks can produce accurate results, so too should the full model. This would not make sense for models that predict actual values, hence we only apply dropout to models that return a probability distribution [8].

This is not such a limiting restriction, though, as classifiers, recurrent neural networks for natural language processing, and many other networks produce probabilistic outputs. With dropout, we are then forcing each node of our network to learn the correct output with less information than it would normally receive. Each subnetwork may then pickup on different features of the input, and shift the parameters in that direction. Applying dropout allows the model to essentially take votes from all subnetworks on the output that should have the highest probability [8]. An example may help to show how dropout can be effective without significantly changing the model output.

Example 5.11. *Suppose we have the network architechture shown in Fig. 5.11. The layers to the left of the diagram are unknown and unimportant as we will simply show the effect of dropping one of the nodes in the second to last layer on the outputs. We will define the values in the second last layer of the network to be* $\begin{bmatrix} 0.5 & -0.2 & 0.7 & 1 \end{bmatrix}^\mathsf{T}$ *and the weights to be given by*

$$\boldsymbol{w} = \begin{bmatrix} 0.6 & -0.5 & 0.3 & 0.9 \\ -0.2 & 0.1 & 0.1 & -0.3 \end{bmatrix}.$$

Then the output of the full model will be $\begin{bmatrix} 1.51 & -0.35 \end{bmatrix}^\mathsf{T}$*, which after applying the softmax function results in a probability of* 86.5% *for the upper node. Now, if the top node in the second last layer was dropped, we can simply change its value to zero in order to see the effect. Doing so results in the new outputs of* $\begin{bmatrix} 1.21 & -0.25 \end{bmatrix}^\mathsf{T}$ *and probability of* 81.2%*. Therefore, dropping a node did not drastically change the output of the model, and this should always be true for fully trained networks. What it has done though, is forced the network to put more importance on the other features represented by the other nodes in the second last layer. The result is that using only that subset of features, the network is not as good at predicting the outcome, hence a different loss and gradient are achieved to account for this.*

Practically speaking, dropout has been shown to work better when applied to larger

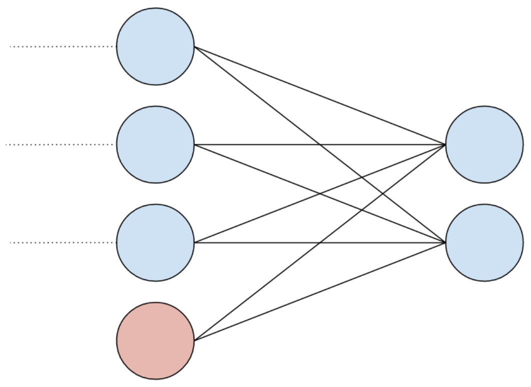


Figure 5.11: A diagram of the last two layers of a network used in Ex. 5.11 to show the effect of dropout on the outputs of a probabilistic network.

networks [8]. Similar to parameter norm penalty regularization, it is also not applied to the bias nodes. For CNNs, it has been empirically shown that applying dropout to the convolutional layers has relatively little effect [17]. Thus, we typically only apply dropout to the fully connected layers of CNNs. For more on dropout, see [8, 17].

Ensemble Networks

The idea of taking output from multiple different networks is not unique to dropout. Ensemble methods are those methods where we train several models, and take the output to be the answer with the most votes. Each different model, even if they all have the same layout, will pick up different features due to different initialization. This works as different models are then likely not to make the same errors when being tested [8]. The advantage of model averaging is that the outputs are only ever as bad as the worst network. So we can only ever increase the accuracy of the model. However, it takes significantly more time and computational power to train the multitude of networks in the ensemble. These models are quite common in competitions to achieve the highest accuracy.

Batch Normalization

The last form of regularization we will cover in this section is called batch normalization. It is an attempt to reduce the effect of a problem that can arise in neural networks called **internal covariate shift** (ICS) [18]. The essence of this problem arises from training the network in batches. As we update parameters of the model that are nearer the input, the parameters closer to the outputs have to deal with a large variety of different distributions. Not only will the parameter updates cause different distributions to occur further in the network, but due to the batches being randomly sampled from the entire dataset, they may have a variety of distributions themselves. The layers further in the network then have to constantly readjust for the different distributions, and the deeper the neural network the worse this problem becomes. Batch normalizaton attempts to overcome this problem by fixing the mean and variance of each batch put through the network.

Suppose the n^{th} layer of our neural network possesses m nodes. Then we can obtain the mean and variance of values in that layer by

$$\mu_B^{(n)} = \frac{1}{m}\sum_{i=1}^{m} x_i^{(n)} \qquad \text{and} \qquad (\sigma_B^{(n)})^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i^{(n)} - \mu_B^{(n)})^2$$

respectively, where x_i is the value in the i^{th} node of the n^{th} layer. Here, we are sub-scripting $\mu^{(n)}$ and $\sigma^{(n)}$ with a B to denote that the values we obtain from these equations are only for batch B of the training dataset. Then we can transform this distribution to one with a mean of zero and variance of one with the mapping

$$\hat{x}_i^{(n)} = \frac{x_i^{(n)} - \mu_B^{(n)}}{\sigma_B^{(n)}}.$$

However, it may not be best for the distribution in each layer to have a mean of zero and variance of one. Forcing this upon the network limits its expressive power, hence batch normalization also transforms each layer as

$$y_i^{(n)} = \delta^{(n)} \hat{x}_i^{(n)} + \beta^{(n)},$$

where $\delta^{(n)}$ and $\beta^{(n)}$ are parameters specific to each layer, and are learned through gradient descent like the other parameters of the model [8, 18]. Note that if $\delta^{(n)} = \sigma_B^{(n)}$ and $\beta^{(n)} = \mu_n$, then this transformation would just return the parameters to their original values. This is possible, and would happen too, but only if they were the optimal values. Additionally, a small number, such as 10^{-8}, is added to the denominator of the first transformation to avoid divisions by zero [8, 18]. Batch normalization has been shown to do more than just attempt to resolve the ICS problem, though. It actually tends to make the landscape of the loss function smoother, stabilizing the gradients, and allowing for faster convergence to the minimum [19].

5.8 Convolutional Neural Network Architecture

Now that we know what neural networks do, and how they work, how should we design a model for a particular task? The answer is relatively unknown. However, certain architectures have been shown to be better than others, and to understand why, we will go through some of the previously winning models of the ImageNet Large Scale Visual Recognition challenge (ILSVRC): A yearly competition in which models must classify images of a database with millions of images and thousands of categories. In the competition, the model must correctly classify as many RGB images as it can, pertaining to a subset of the full dataset that spans $1,000$ categories. These images vary in size, however, they are typically preprocessed to $3 \times 256 \times 256$, where the three represents the colour channels. Due to the size of these images, a CNN should be used, as a fully connected network would be too large, and

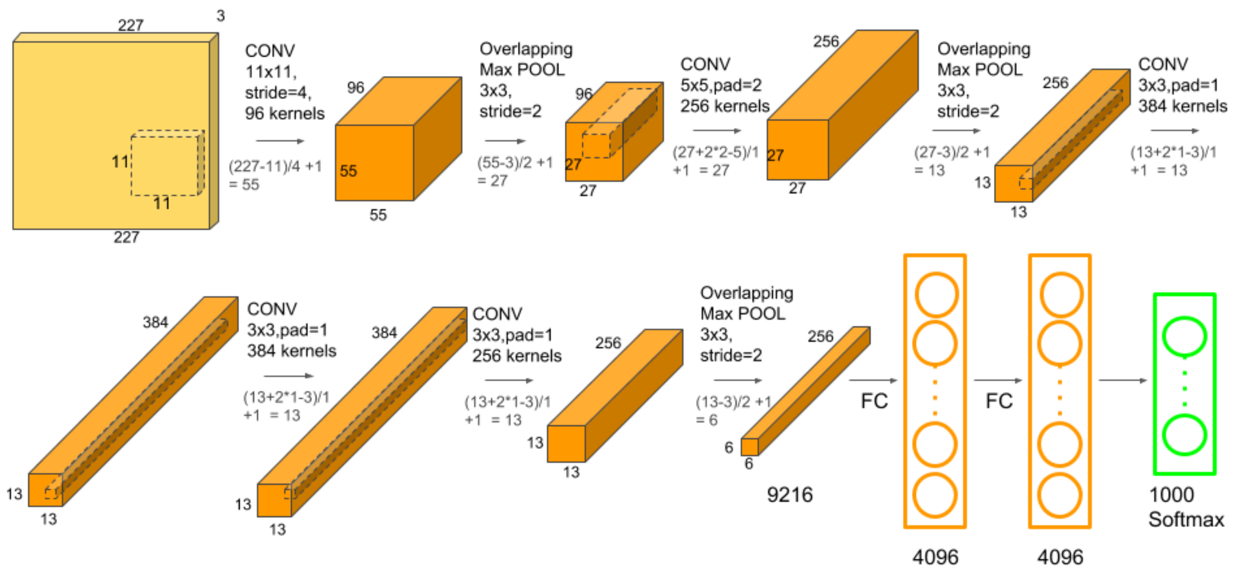


Figure 5.12: This figure displays the structural design of AlexNet, the 2012 ILSVRC winner. The height and width of the convolutional layers represents the size of the feature maps, and the depth represents how many kernels were applied to the previous layer. Found from: https://www.learnopencv.com/understanding-alexnet

take up too much memory.

AlexNet

AlexNet is the name of the network that won the competition in 2012. The property that set it apart from the other networks is that it was much larger than any other convolutional network that had been seen previously. In fact, it was so large that it had to be split over two different graphics processing units (GPUs); the hardware that is typically used to train neural networks, as they perform many simple computations very quickly. AlexNet had the architecture shown in Fig 5.12.

What we can learn from this configuration is that we should slowly decrease the size of the feature maps, performing at least one convolutional layer between the pooling layers.

Then, once the dimensions of the feature maps are small enough, switch to fully connected layers. The other take-away from this model is that it is good to have lots of kernels. While we do need to keep track of the number of parameters of our network to ensure it does not get too large, each new kernel will detect a different feature and can thus be a very useful addition to the model. For more on this model see [20].

VGGNet

Next we will look at the runner-up for the ILSVRC 2014 competition. A common way to measure the accuracy in this competition is whether the true category is among the top five predicted classes. It was in this year that this top-five error rate reached values below 10%. The runner-up for the 2014 competition was called VGGNet, a network similar to that of AlexNet. The reason we are examining this network is that it is scalable, and performs quite well at each level. In fact, it was not just VGGNet that was the runner-up in 2014, but each version of the network took all top places aside from first. A visual representation of each of the variations of this network can be seen in Fig 5.13.

There are several things to note about this network. First of all, it not only has more kernels per convolutional layer than AlexNet, but it also has more convolutional layers in general. The theme that we start to see, is that as we decrease the height and width of the feature maps, we increase the depth, or number of feature maps. Along with having more convolutional layers, the kernel sizes in each layer are smaller than that of AlexNets. This is another parameter saving method.

Consider the following two scenarios:

1. One 7×7 convolutional layer
2. Three sequential 3×3 convolutional layers

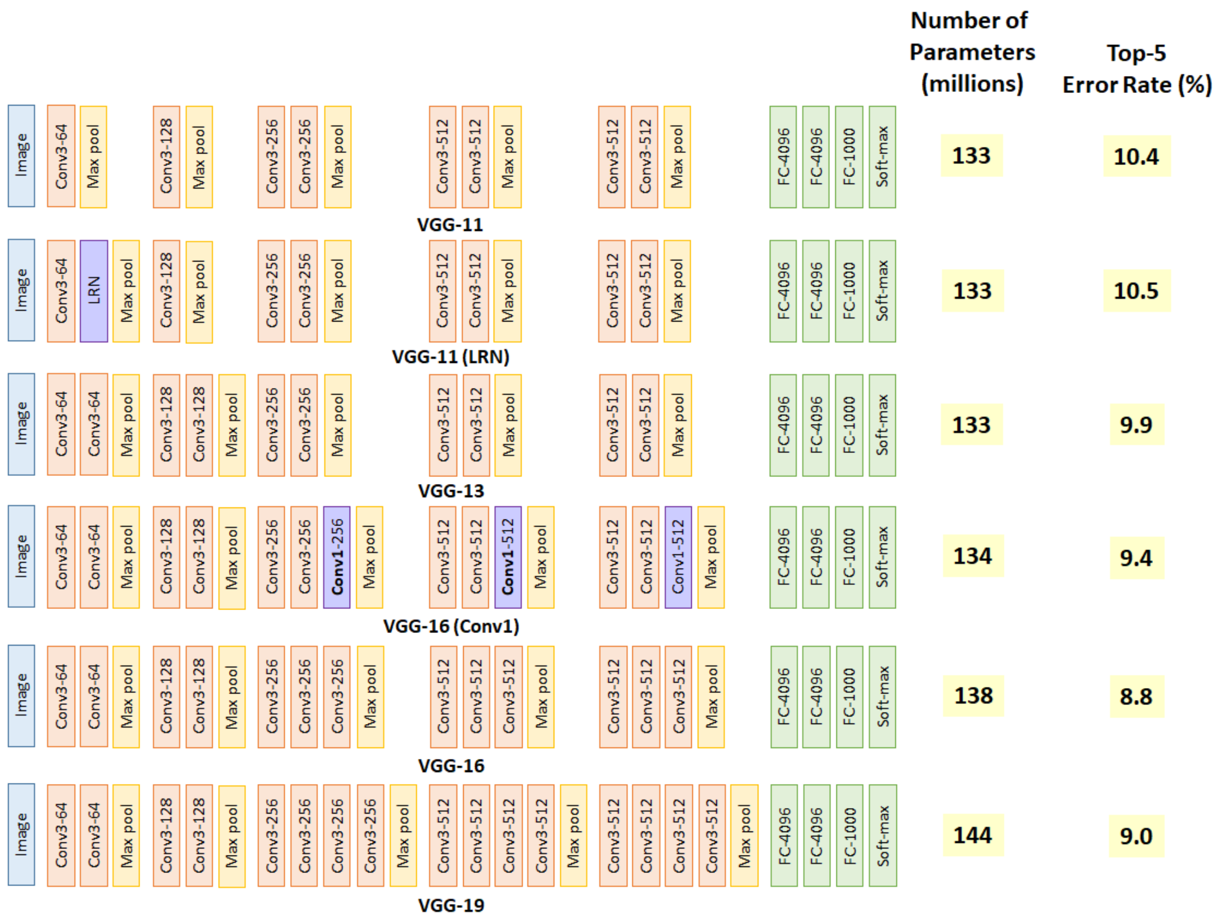


Figure 5.13: This image shows the architecture of each of the VGGNet variations, the runner-ups for the ILSVRC 2014 competition. In this graphic, Conv3-64, for example, represents a convolutional layer with 64 3 × 3 kernels. Found from: https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

In the first scenario, each value in the feature map represents one 7×7 block of the image, and there are a total of 49 parameters involved to calculate it. In the second scenario, the feature map after the first layer will be composed of values that represent a 3×3 block of the preceding image. Then, in the output from the second layer, each value will represent a 5×5 block from the original image. After the last filter, each value in the feature map will represent a 7×7 block of the original image, matching that of the first scenario. However, the second option only has 27 parameters as opposed to the 49 of the first. Hence, in terms of reducing the number of parameters of a model, it is better to have more layers with smaller kernels.

Another aspect one may notice is that there are special purple blocks in this figure. The LRN block represents a local response normalization layer. This is similar to batch normalization, but in a localized area of the feature map. This type of layer will not be used, hence we will not discuss it further. The other purple block has a 1×1 kernel. This may lead one to think that it is an unimportant layer, however, since filters have different weights for each feature map, we can think of this as selecting and adding specific weights of some feature maps the model deems useful. In addition, 1×1 convolutions can be used to change the dimensionality of the its input in order to reduce the number of parameters in the model.

Example 5.12. *In this example we demonstrate how* 1×1 *convolutions can reduce the number of parameters involved in a convolutional layer. Specifically, since each kernel, no matter the number of input feature maps, produces a single output feature map, we can use* n 1×1 *filters to reduce the number of feature maps on which we perform other calculations. Suppose we take as input an object of size* $256 \times 64 \times 64$*, and we would like to apply* 256 *filters of size* 3×3*. We have the following two methods by which we can accomplish this task:*

1. *Directly compute the* 3×3 *convolutional layer with* 256 *filters.*
2. *Reduce the number of feature maps to 64 by applying 64* 1×1 *filters, compute* 64 *feature maps using* 3×3 *kernels, and re-expand the number of feature maps by applying 256* 1×1 *filters.*

If we take the first option, then the total number of parameters involved will be

$$256 \times 3 \times 3 \times 256 = 589,824.$$

However, the number of parameters involved in the second scenario would be

$$256 \times 1 \times 1 \times 64 + 64 \times 3 \times 3 \times 64 + 64 \times 1 \times 1 \times 256 = 69,632.$$

Thus, we have reduced the number of parameters involved in the calculation by almost a factor of ten!

For more on the VGG network see [21].

GoogLeNet

The network that beat VGG, and the winner of the competition in 2014, was designed by Google, and is called GoogLeNet, or Inception-v1. It is with this model that we get the idea of blocks in a neural network. The motivation behind the blocks, or in this case called inception modules, is that different filter sizes may pick up different features. However, there is no reason we need to choose between the filter sizes. We can see the design of the inception modules in Fig. 5.14, and the full layout of GoogLetNet in Fig. 5.15.

The inception module fully takes advantage of the dimensionality reduction provided by the 1×1 convolutions, as shown by the purple blocks in Fig. 5.14. We can then also choose

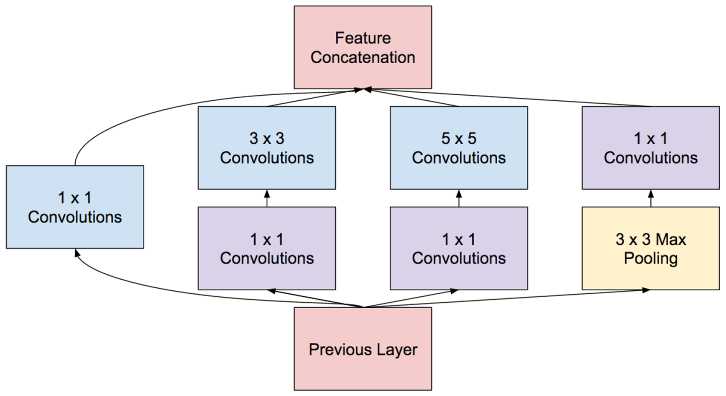


Figure 5.14: The inception module, or block used in GoogLeNet.

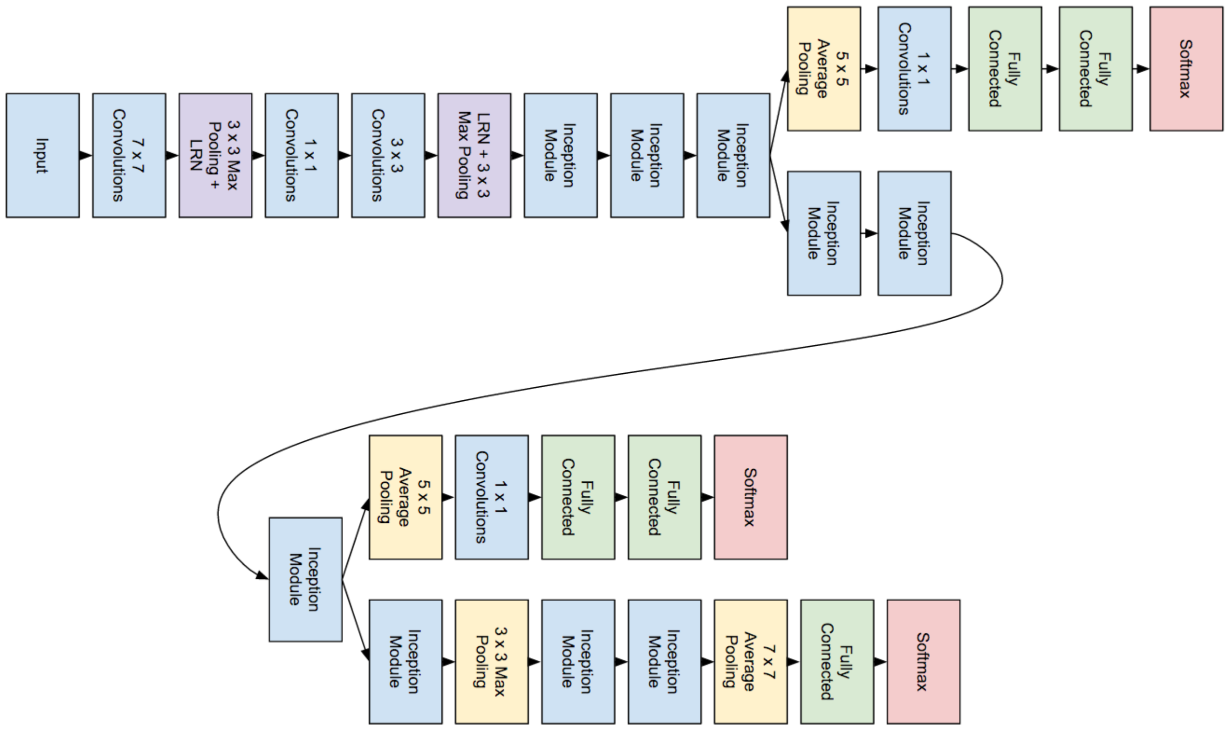


Figure 5.15: The architecture of GoogLeNet, the 2014 ILSVRC competition winner. Each inception module block performs the computations shown in Fig. 5.14.

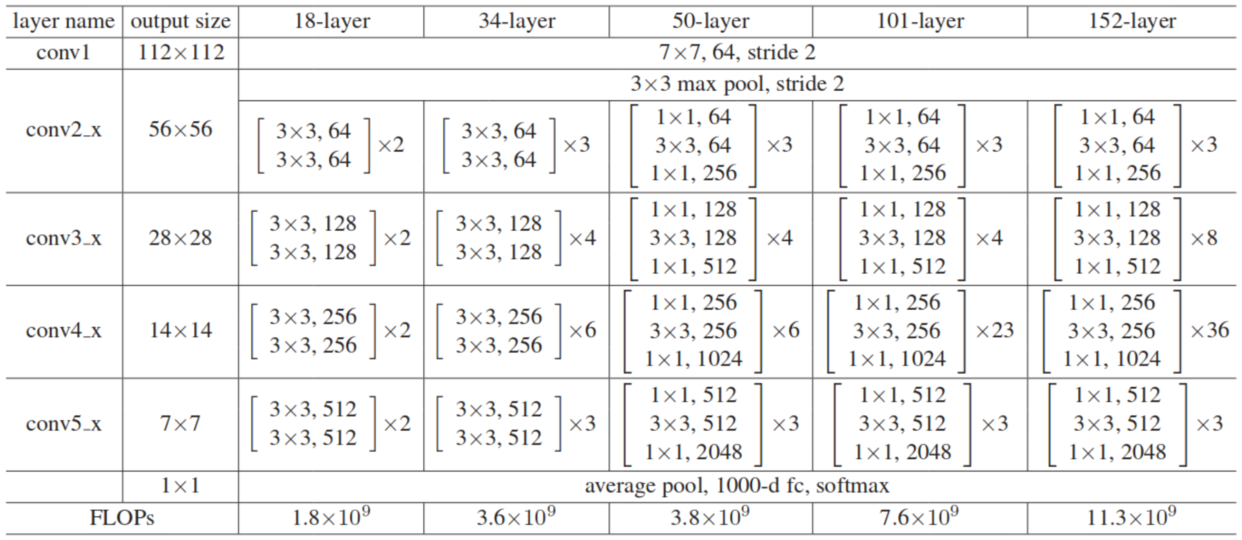
the number of features we want in the concatenation at the end of this block, and more specifically, how much of each convolution we want represented. This allows the network not to have to choose between the filter sizes but instead take the important components from each one.

As for the full GoogLeNet model itself, you may notice that it is somewhat strange looking, and has multiple end points (the red blocks). The softmax function at the end of the longest chain is the true end of the network, though. The intermediary ends of the network are used to help the network get through the ever-present vanishing gradient problem. Although the team behind GoogLeNet used the ReLU activation function between the layers, very long networks such as this are still prone to vanishing gradients. For more on this network see [22].

ResNet

The last network that we will examine in this section is the 2015 ILSVRC winner, called ResNet. This network is not only important for its concept, but because it was the first model to surpass human-level performance in the ILSVRC. In 2014, after training for a few days, Andrej Karpathy tested himself in the ImageNet competition and obtained a top five error rate of approximately 5.1%. While this may initially appear to be a poor accuracy for human performance, with $1,000$ classes, the categories can become quite definitive and delve into specifics such as breeds of dogs.

The key concept to ResNet is another method to overcome the vanishing gradient problem. That is, we can create a "gradient-highway" by including **skipped connections**. Skipped connections are where the output of a node is not only connected to the next layer, but layers further along the network as well. Many different lengths of the ResNet model were tested, their designs are shown in Fig. 5.16. The ResNet-152 model produced the best



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 5.16: The design of each of the ResNet models used in the 2015 ILSVRC competition. Each of the blocks contained within square brackets posses a skipped connection between the start and end of the block. Inside the square brackets shows the filter size followed by the number of feature maps created from that filter.

results.

The effect of these skipped connections is what gives ResNet its name. Let x be the input to a given layer and $T(x)$ be the computations that layer, or multiple layers would perform if there were no skipped connections in the model. Then the difference between the input and the output, or the residuals is given by

$$R(x) = T(x) - x.$$

Hence by rearranging, we see that if we add the input of a layer to the output of the following computations, the network during training now approximates $R(x)$;

$$T(x) = R(x) + x.$$

These skipped connections help to overcome the vanishing gradient problem through the

Model	Top-5 Error (%)	Number of Layers	Number of Parameters ($\times 10^6$)
AlexNet	15.3	8	62
VGG-16	8.8	16	138
GoogLeNet	6.67	22	6.8
ResNet-152	3.57	152	62

Table 5.3: A comparison of all the networks discussed in this section. This table compares the top five error percentage, number of layers, and number of parameters in each of the models.

shorter path they create. Overcoming the vanishing gradient problems allows for much deeper networks to be created. For more on ResNet, see [23].

A comparison of performance, number of layers, and number of parameters of all the models discussed in this section can be found in Table 5.3. The theme in this table is that we are creating longer networks that perform better, while simultaneously attempting to minimize the number of parameters. Since the ILSVRC 2015 competition, there have been newer models with better accuracy, such as ResNext [24], Inception-v4, or Inception-ResNet [25], and several others. Many of these networks are modified versions of GoogLeNet and ResNet. We will not use these models, so they will be omitted.

Chapter 6

Predicting Iterated Function System Parameters

In this chapter we combine the areas of fractals and neural networks, and attempt to provide a solution to a long standing inverse problem: Given an image, can we obtain parameters of an IFS that has an attractor resembling that image? Solutions to this problem have been found for several images that were already attractors, however, this was the result of exhaustive searches, see [3, 4, 5, 6, 7] for details on these. We would like to solve a more general version of this problem: Obtain a mapping from an image to IFS parameters pertaining to an attractor resembling that image. Throughout this chapter we will construct a neural network model that will approximate this function, train it on the fractal databases constructed in Chapter 3, and provide samples of output from the model, as well as model accuracies.

6.1 Designing The Model

The goal of this section is to incorporate many of the methods of the previous chapter in order to design a network that, when trained on the fractal databases we created, predicts parameters of an IFS that has an attractor resembling a given image. We will mimic the structure of the ResNet model as it has been shown to perform well and allow for very deep neural networks. However, ResNet was originally used for classification, made use of the CE loss function, and had a manually adjusted learning rate. Since we want our model to predict the parameters of an IFS we will modify the ResNet structure, and incorporate the following features:

- The Adam optimizer.
- Batch normalization between each layer.
- The Leaky ReLU activation function between each layer.
- $6N$ outputs corresponding to the parameters of the N functions in each IFS of the database the network is being trained on.
- The MSE loss function.
- L2 weight decay.

As shown in the Convolutional Neural Network Architecture section, the ResNet architecture relies on blocks created from multiple convolutional layers with a skipped connection. The block configuration from which we will construct our network is shown in Fig. 6.1, where the size of the kernel in the middle layer will be treated as a hyperparameter. Note that in order to be able to add the input of the block to the output produced from the convolutional layers, their dimensions must match. Thus, the convolutional layers must preserve these

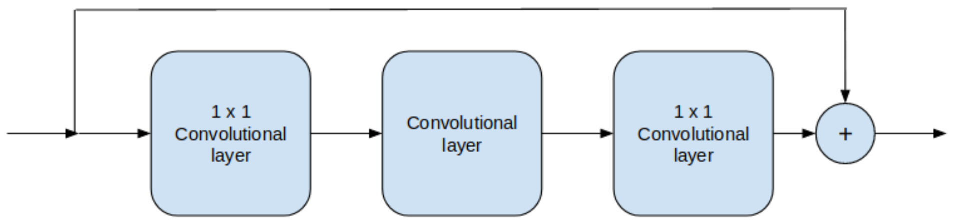


Figure 6.1: The block structure that we will use in our model. The size of the filter in the middle layer will be treated as a hyperparameter.

dimensions. The 1×1 convolution layers will do this automatically, however, the middle layer will require zero padding.

Each of the convolutional layers shown in Fig. 6.1 is followed by batch normalization and the Leaky ReLU activation function. The first layer of convolution in each block is used to reduce the number of feature maps from the input, whereas the last layer expands it back to the original amount. We will call the uninterrupted repetition of blocks in a network a **chunk**. That is, a chunk of size three corresponds to three repeated blocks. Between each chunk in our network are pooling layers. We chose to make each pooling layer have a stride of two, where the size of the kernel became a hyperparameter. The kernel size was always chosen to be an even number, k, so that $k/2-1$ layers of zero padding allowed for the height and width of the image to be perfectly cut in half. We additionally chose to double the number of feature maps each pooling layer took as input.

Since the size of the images in our databases are significantly larger than those in the ImageNet database, we will require more pooling layers than ResNet to reduce the dimensionality sufficiently. With the chunks and pooling layer framework in place, we sequentially computed a pooling layer followed by a chunk a total of seven times. With an initial pooling layer resulting in 8 feature maps of size 320×320, the output from the last layer would then be an object of size $512 \times 5 \times 5$. This was followed by one or more fully connected layers leading to the outputs.

The next step in creating the model is to fine-tune the hyperparameters. Note that the depth of the model is now controlled by the size of each chunk, taking its place as a hyperparameter. This leaves the following hyperparameters we must adjust:

- The number and width of fully connected layers.
- The size of each chunk.
- The size of the filter in each block.
- The size of the filter in each pooling layer.
- The learning rate.
- The momentum coefficients.
- The size of each batch.
- The regularization coefficient.

The momentum coefficients were kept at their defaults as Adam has been shown to be robust to many applications with respect to hyperparameters other than the learning rate [8, 16]. These defaults were 0.9, and 0.999 for the momentum in the parameter update and learning rate scaling factor, respectively. Additionally, the L2 weight decay factor was kept at its default of 0.01 as well, since we have large enough datasets that overfitting should not be an issue.

All models were written in version 3.7 of the Python programming language with version 1.2 of the pytorch machine learning library. They were each trained on two GPUs in the Graham cluster of the compute Canada resources. This allowed for a total of 12GB of memory for the model. Due to the size of our images, this leaves relatively little space for the fully connected layers, so only two configurations were tested. After flattening the $512 \times 5 \times 5$ object to a vector of size 12,800, the fully connected layers either

1. Connected to a layer of width 500, followed by a layer of width 100, followed by the $6N$ outputs. Or,

2. Connected to a layer of width 1,000, followed by the $6N$ outputs.

Due to the memory limitations, there was also only a limited number of chunk sizes possible as well. A scheme similar to ResNet-152: Chunk sizes of 3, 8, 36, and 3, scaled to seven chunks, would be ideal to test, but not possible. This is also the reason the number of feature maps from our initial pooling layer is 8, as opposed to the 64 found in ResNet-152. With this in mind, if our first chunk is of size 3, then the second must be of size 4, or 5, should we maintain the concept of having much larger chunks near the end of the convolutional layers, similar to ResNet.

In order to find near-optimal hyperparameters, we performed a grid search of those remaining. The model was trained for two epochs with a given configuration and set of hyperparameters, and then tested for its accuracy to discriminate which combination is best. The accuracy of the model was tested using various tolerance levels. If a value output by the model is within the given tolerance of the corresponding true value, it is considered correct.

Example 6.1. *Suppose we gave the model an attractor corresponding to the IFS*

$$W = \begin{cases} w_1(x,\, y) & = \begin{bmatrix} 0.387439 & 0.032326 \\ -0.509266 & -0.686624 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.209960 \\ 0.369290 \end{bmatrix} \\ w_2(x,\, y) & = \begin{bmatrix} 0.442961 & -0.458575 \\ -0.166075 & 0.588832 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.355568 \\ -0.350048 \end{bmatrix} \end{cases},$$

Tolerance	0.1	0.2	0.3	0.4	0.5
Accuracy	4/12	9/12	10/12	11/12	12/12
Accuracy (%)	33.33	75	83.33	91.67	100

Table 6.1: Calculating the accuracy of a neural network at various tolerance levels

and it produced parameters corresponding to the IFS

$$W = \begin{cases} w_1(x,\, y) &= \begin{bmatrix} -0.034522 & 0.078493 \\ -0.495476 & -0.455910 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.285416 \\ 0.245609 \end{bmatrix} \\ w_2(x,\, y) &= \begin{bmatrix} 0.537892 & -0.272841 \\ -0.273395 & 0.281454 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.233959 \\ -0.247660 \end{bmatrix} \end{cases}.$$

Then the accuracies it would receive for this particular piece of data are given in Table 6.1

Alternatively, we could have measured the accuracy based on the similarity between the input and the attractor of the model produced IFS. However, fractals are typically quite sensitive to their parameters; a small change in a few parameters of a single function can perturb the attractor of an IFS a significant amount. This concept, paired with the fact that none of the models had a high accuracy under the method shown in Ex. 6.1, means that measuring the difference in attractors would not be an accurate measurement to discriminate between sets of hyperparameters.

The effect of the structure of the network on the model accuracy was found to be fairly consistent throughout each database. This gives us the freedom to use the same model architecture with different hyperparameters for each dataset, allowing for a direct comparison of the models after each one is trained. Of the various schemes of chunk sizes tested, they all produced relatively similar results. However, it was still beneficial to have an increase in the size of the chunks near the end of the convolutaional layers. In addition, the larger models performed slightly better. The model design found to work best, and that we will train on

Functions in the IFS	2	4	6	8
Learning rate	7.5×10^{-6}	7.5×10^{-5}	5×10^{-5}	2.5×10^{-5}

Table 6.2: Learning rates chosen for the neural networks used to predict IFS parameters of each database.

each of the databases, has chunk sizes of 3, 5, 7, 11, 29, 23, and 17.

The fully connected layers impacted the accuracy of the model more significantly than the chunk sizes. The first scheme; the one that was not chosen, was typically $2 - 3\%$ less accurate at each tolerance level. This makes sense as it has less parameters than the second fully connected layer option. Additionally, the filter sizes in each block and pooling layer had a significant effect on the accuracy as well. Larger filter sizes reduced the accuracy of the model, on average, by up to 2%. This is thought to be due to the additional zero padding required to maintain the dimensions of the input, allowing each block to approximate its residuals. Thus, filter sizes of three in each block, and four in each pooling layer were chosen.

Despite the architecture of the model being invariant among the databases, different learning rates were found to work best for each of them. Learning rates of the form 10^{-m} where $m = [1, 2, 3, 4, 5, 6]$ were tested. Upon testing these, a bisection method of sorts was performed between the top two values for two iterations, until the selected value was found. Performing further bisections was not found to impact the accuracy significantly. This resulted in the learning rates shown in Table 6.2

A batch size of 100 was chosen for the purpose of it being the largest even number that is a divisor of the total dataset size. The number being the largest possible was chosen to reduce the training time through the multiprocessing capabilities of the GPU, which already takes approximately an hour and a half per epoch. An even number was required because of how pytorch implements the use of multiple GPUs. Essentially, two identical models are created, and stored on the separate GPUs, each possessing half the batch. After feeding the data through the network, the outputs are concatenated and treated as one.

To summarize, our models have the following hyperparameters:

- A fully connected layer of width 1,000 between the output of the convolutional layers and the model outputs.
- Chunk sizes of 3, 5, 7, 11, 29, 23, and 17.
- Filters of size 3 in each block.
- Filters of size 4 in each pooling layer.
- Learning rates shown in Table 6.2.
- A momentum coefficient of 0.9 in the parameter update, and 0.999 in the learning rate scaling factor
- A batch size of 100.
- A regularization coefficient of 0.01.

6.2 Model Training, Analysis, And Performance

In order to fully train the models, we used the entire previously generated datasets of 250,000 images as the training set for each model. A validation set of size 10,000, and a testing set of size 10,000 were subsequently created and used to assess when to stop training, and measure the accuracy of the model. Despite the implementation of the Adam optimization algorithm, it was found that the learning rates had to be manually adjusted throughout the training process. In order to modify the learning rate at the appropriate time, the model was saved after each epoch of training. Then, when the loss increased, or the validation accuracy

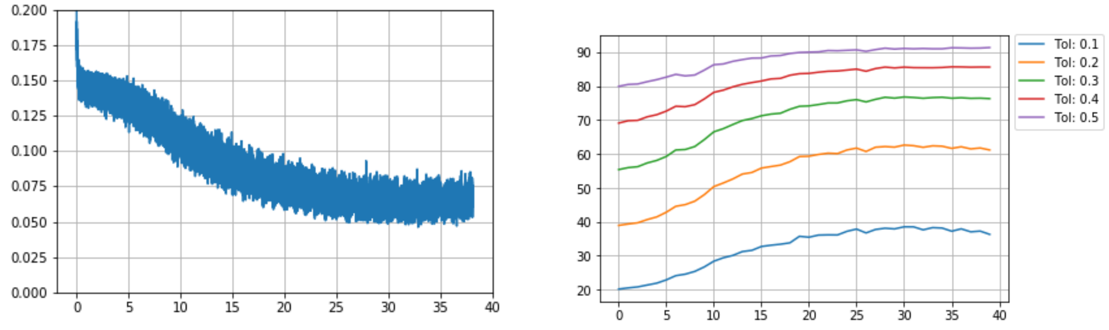


Figure 6.2: The loss and validation accuracy of the model trained on IFSs possessing two functions. The x-axis for both of these images is the epoch, and the y-axis is the loss and accuracy percentage for the image on the left and right, respectively.

decreased for several epochs, we returned to a previous state of the model and decreased the learning rate by a factor of ten.

Example 6.2. *The clearest depiction of when the learning rate had to be adjusted is found in the model for two-function IFSs. The loss and validation accuracy throughout training are shown in Fig. 6.2. As we can see from these images, the loss of the model begins increasing, and the accuracy of the model begins decreasing at approximately epoch* 32*. Hence at this point in the training process, we returned to epoch* 32*, decreased the learning rate by a factor of ten, and kept training. This allowed the model to progress further.*

The learning rate had to be decreased at different epochs for each model; these epochs can be found in Table 6.3, along with the total number of epochs for which each model was trained. Note that the first time the learning rate was scaled for the two-function IFS model is much later than the others. This could in part be due to the fact that the initial learning rate was much smaller, though it could also be a result of the distance from the network parameter initialization to the found solution. To go with this information, graphs of the loss and validation accuracy of each model throughout the full training period are given in Fig. 6.3.

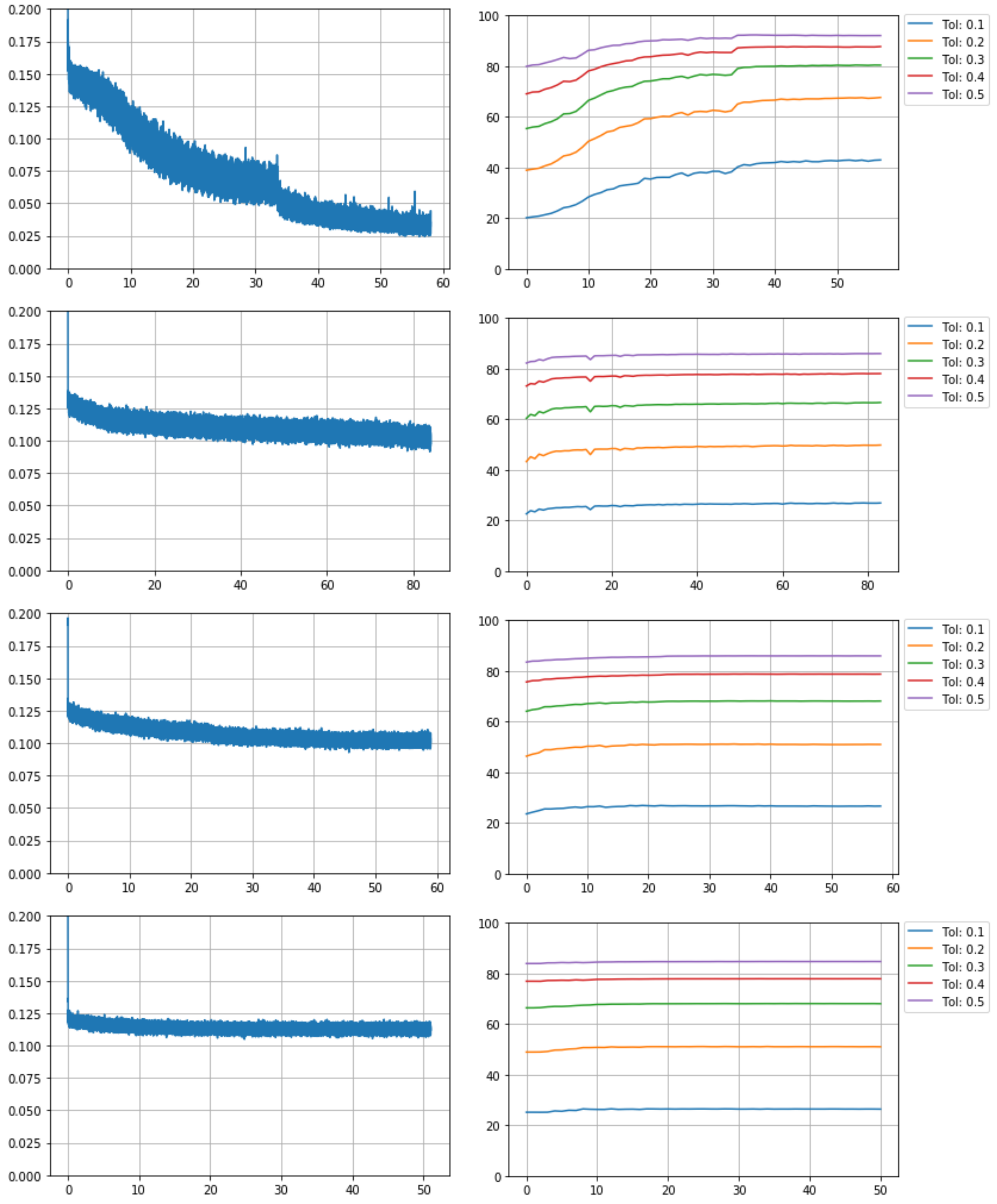


Figure 6.3: The loss and validation accuracy of each model during training. Going from top to bottom, each row corresponds to the model trained on IFSs possessing two, four, six, and eight functions, respectively. The x-axis for all images is the epoch, and the y-axis is the loss and accuracy percentage for the images on the left and right, respectively.

Functions in the IFS	2	4	6	8
Learning Rate Decrease Epoch	33, 54	6,77	3, 23, 42	9, 17, 33, 42
Total Epochs	58	84	59	51

Table 6.3: Here we show for each model at which epochs the learning rate was decreased by a factor of ten, and the total number of epochs for which the model was trained.

Indeed we did not train the models to zero loss. Instead, we stopped the learning process whenever either the accuracy began to diminish, the loss began to increase, or a combination of both occurred, and decreasing the learning rate had relatively no effect.

There are several other aspects for each model that we can analyse from the graphs in Fig. 6.3. To begin with, the initial accuracy of the model for each database increased the more functions there were in the IFS. This is likely not a cause of the model, but the data. Specifically, due to the scaling of the fractals, the distribution of the additive parameters; those not part of the matrix in the affine transformation, did not follow a normal distribution. Examples of these distributions are shown in Fig. 6.4. We can see from these images that on average, the magnitude of these parameters decreases with more functions. Hence, a neural network attempting to predict these parameters can do much better on IFSs consisting of more functions by simply producing a smaller value.

Despite the increase in initial validation accuracy, it is seen in Fig. 6.3 that the more functions in the IFS, the less the loss decreased, and validation accuracy increased, throughout training. This is likely due to both the data, and the model complexity. For instance, since the model complexity is constant, apart from the last fully connected layer leading to the outputs, it makes sense that it would not model a function mapping an attractor to an eight-function IFS as well as it would to a two-function IFS. This is because the complexity of the attractor itself is increased as there are more parameters required to create it; empirical evidence of this is given at the end of this section. The data plays a role in this effect as well. Recall that as we increase the functions in an IFS, we go from obtaining attractors

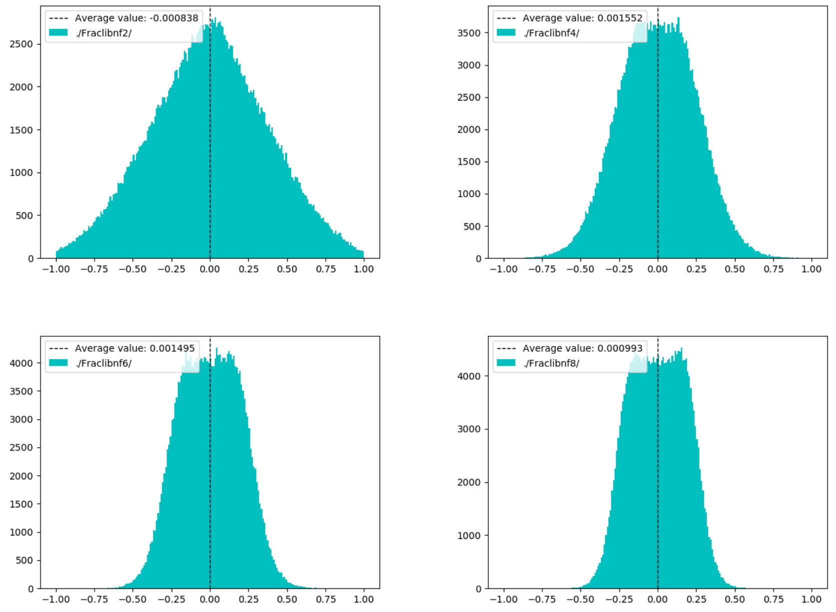


Figure 6.4: Distribution of the additive parameters of an IFS. The top-left, top-right, bottom-left, and bottom-right correspond to the databases of IFSs consisting of two, four, six, and eight functions, respectively.

represented by fewer pixels that are more spread out, to ones with more pixels that are closer to the fractal centroid. This essentially means that, on average, those attractors with more functions in their corresponding IFS are more similar to one another than those generated from an IFS with fewer functions. Hence, it is more difficult for the neural network to distinguish important features, and requires a more complex model.

The final model accuracies at each tolerance level averaged over their entire respective 10,000 fractal testing sets are given in Table 6.4. These accuracies quantify numerically the difference between the model output and the true parameters, however, they do not necessarily reflect the similarity of the attractor predicted by the model to that of the input. Thus, we will reconstruct the attractors of the IFSs produced from the model. We cannot reconstruct just any output that the model calculates, though. We must once again ensure that the functions are contractive. That being said, after calculating the model outputs for

Tolerance	Accuracy (%)			
	2 function IFS	4 function IFS	6 function IFS	8 function IFS
0.1	43.17	26.74	26.53	26.46
0.2	67.96	49.52	50.76	51.04
0.3	80.81	66.60	68.07	68.06
0.4	87.92	78.10	78.83	78.00
0.5	92.26	86.02	85.99	84.79

Table 6.4: The accuracy of each model at various tolerance levels

the testing set, almost all of them are contractive. This may be attributed to the fact that the set of matrices A, such that $||A||_2 < 1$, is convex. Since each model was trained only on values within this convex set, it follows that it should mostly produce values within it as well. Additionally, note the the attractors of the model produced IFSs are not necessarily in $[-1, 1]^2$. Despite this, we kept the viewing region as $[-1, 1]^2$ so as to ensure consistency when comparing with the input.

In Fig. 6.5 we compare the model input to the attractor of the model produced IFS for an element in each test set. The images chosen were characteristic of many of the reconstructions. As we can see, the model produced IFSs possess quite sparse attractors, though for fewer functions in the IFS, this is more a feature of the dataset rather than the model. Another attribute represented in these images is that each of the models displayed the ability to match the centroid of the input quite well.

In Fig. 6.6 we show some of the attractors of model produced IFSs compared to their input where the model fractals were selected from some of the least sparse outputs. We can see that the two and four function IFS models retain some of the features of the input attractor. While there is only a handful of non-sparse attractors that possess this attribute for the four function model, the network trained on IFSs consisting of two function has numerous such cases, for more, see the appendix. Additionally, notice that as the number of functions in the IFS increases, the sparsity of the least sparse model produced attractors

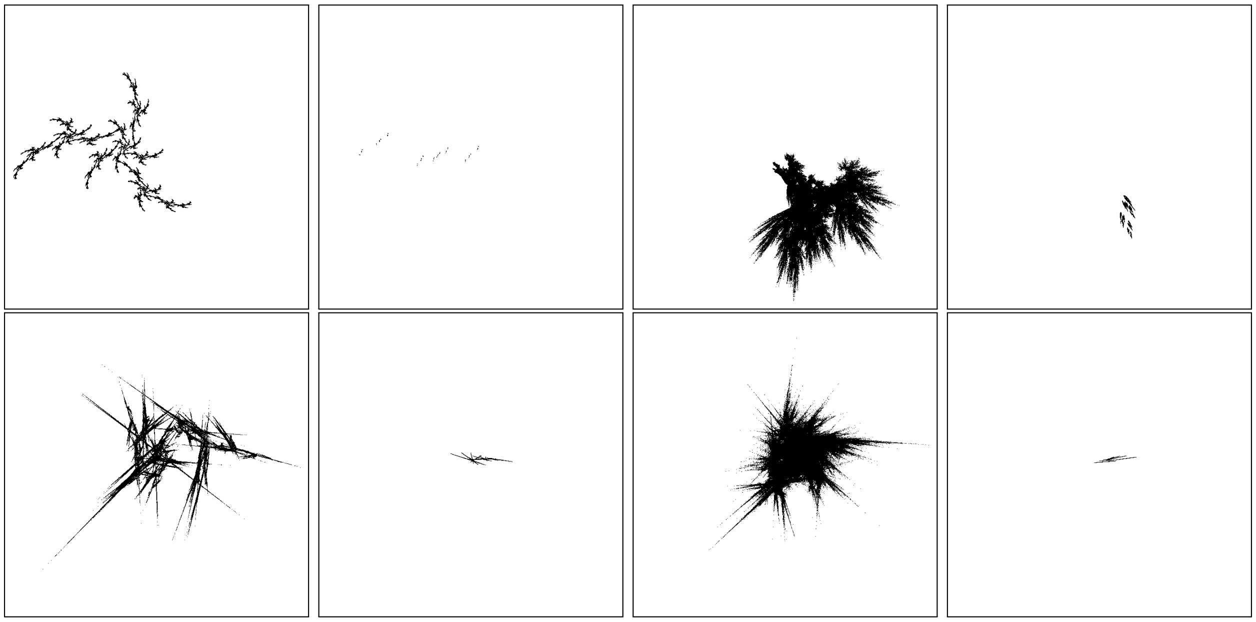


Figure 6.5: Attractors of model produced IFSs compared to the model input. The first and third column are the are images of the input, and to their right are the model produced IFS attractors. The top-left, top-right, bottom-left, and bottom-right input and reconstruction correspond to the two, four, six, and eight function models respectively.

increases. This can likely be attributed to the distribution of the additive parameters once again, as well as insufficient model complexity. Since the model was only trained on those IFSs that have small additive parameters, it will likely only produce IFSs with small additive parameters. See the future work chapter for a possible method of overcoming this.

A question that one may ask with a mapping from an attractor to IFS parameters is whether or not there exists two similar attractors that correspond to IFSs with an unequal amount of functions. Further, whether or not one can approximate non-fractal sets with this mapping. Fig. 6.7 provides examples of applying the two-function IFS model to images of the other databases, as well as an image that is not a fractal. We only show results for the network trained on IFSs consisting of two functions as this network approximated the true mapping from an image to IFS parameters best. Additionally, note that the images in Fig. 6.7 are selected from the lowest sparsity reconstructions, and do not reflect how well

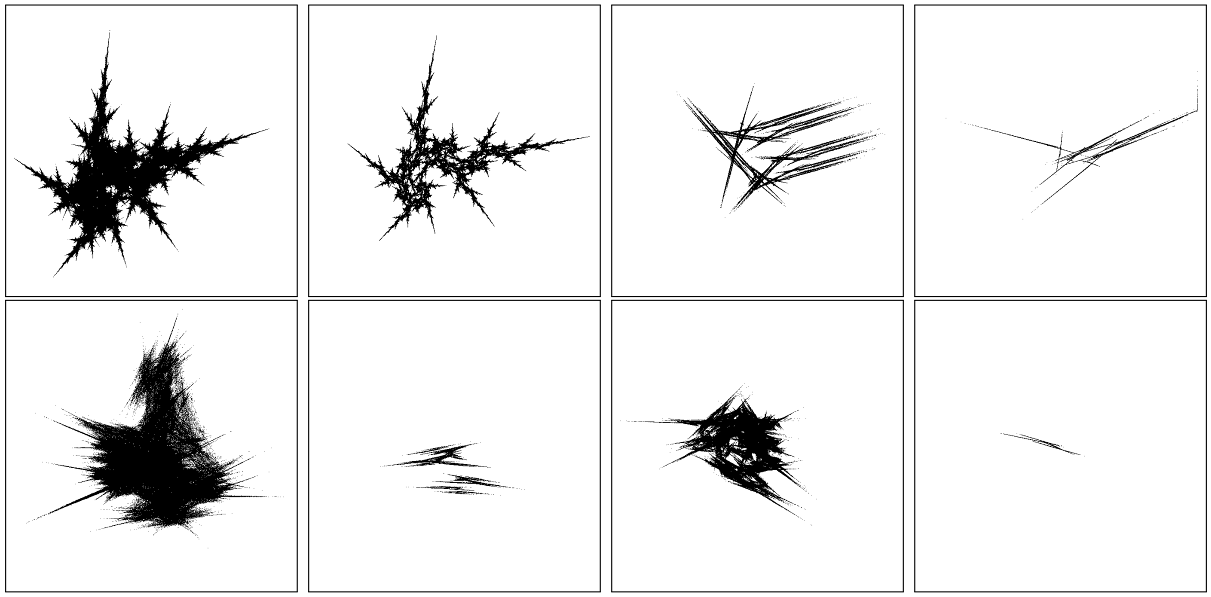


Figure 6.6: Some of the least sparse attractors of model produced IFSs compared to their model input. The first and third column are the are images of the input the produced the IFS with the attractor to its right. The top-left, top-right, bottom-left, and bottom-right input and reconstruction correspond to the two, four, six, and eight function models respectively.

it approximated each fractal from the other databases. However, it does give an estimate of the best the model can approximate these alternative inputs. As we can see, the model produced attractor becomes a worse approximation of the input, the greater the number the functions in the input attractors IFS. For the non-fractal image, the approximation is yet worse. Thus, this is empirical evidence that attractors created from IFSs with a greater number of functions have a greater complexity in the sense that it requires more parameters to model them. Non-fractal images are then akin to the attractor of an IFS with a very large amount of functions.

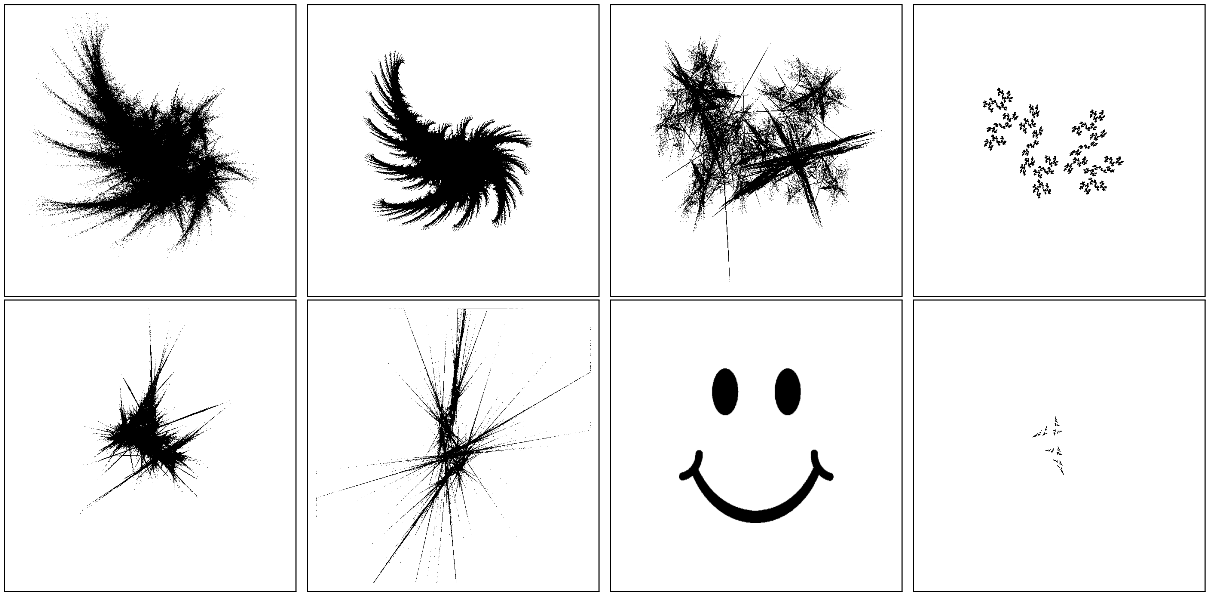


Figure 6.7: Some of the least sparse attractors of the two-function IFS model when the input is not an attractor from a two-function IFS. The first and third column are the are images of the input that produced the IFS with the attractor to its right. The top-left, top-right, and bottom-left, input and reconstruction correspond to attractors with four, six, and eight functions, respecitvely, and the bottom-right is an example of inputing a non-fractal image.

Chapter 7

Future Work

We restricted ourselves at many points throughout this thesis, reducing the areas we explored in both IFSs constructed from nonaffine functions, and neural network implementations of IFS parameter estimation. Additionally, there are many extensions from this work that could be implemented. Here, we briefly describe some of these methods.

IFSs Constructed From Nonaffine Functions

When exploring the area of IFSs comprised of functions with bounded derivatives, we restricted ourselves to transformations in which $g_1 = g_3$ and $g_2 = g_4$. Relaxing this restriction could produce interesting results. Additionally, there are many more bounded derivative functions that could be tested. For instance, the function $g = \sqrt{|x| + a}$ has a derivative bounded by

$$M = \sup_x |g'(x)| = \lim_{x \to 0} \left| \frac{x}{2|x|\sqrt{|x| + a}} \right| = \frac{1}{2\sqrt{a}}.$$

Many interesting methods of colouring fractals have been found. One could apply these to both IFSs constructed from bounded derivative functions, and piecewise IFSs.

There are many concepts left untested for the piecewise affine IFSs as well. More complex piecewise boundaries, such as $\sin(ax) = b$, $cos(ay) = b$, or some combination such that $a \in \mathbb{R}$, $b \in (0, 1)$ could produce interesting results, particularly for $|a| > 1$. One could also implement more than two branches for an IFS, or construct an IFS where each function has a different piecewise boundary. All of these have unknown results.

For both types of alternative functions we only examined those IFSs constructed from four functions. One could generate databases, as we did for affine functions, and examine some properties of these IFS attractors as the number of functions is varied.

Another possibility with piecewise IFSs is to use nonaffine transformations in each branch. That is, construct a piecewise IFS from bounded derivative functions. One could also have one branch being a bounded derivative transformation, while the other is affine. Similarly, one could construct an IFS possessing any combination of affine functions, bounded derivative functions, and piecewise functions.

There were many concepts left unexplored in terms of piecewise functions as well: Is the change in an attractor set continuous for continuous changes in the piecewise boundary? That is, if the piecewise boundary is defined by $x = a$, for example, and a is varied continuously, how does the attractor change?

In terms of fractal splicing, there are some additionally untouched ideas. When splicing two IFSs with unequal amounts of functions, how does the attractor change when varying which functions are kept affine? One could also explore more specific boundaries with fractal splicing. In other words, splice two fractals with a piecewise boundary specifically defined so that only particular regions of each branches attractor crosses the boundary.

Bounded Range Functions

During the explorations detailed in section 4.1, a similar exploration was performed on functions with a bounded range; that is, functions satisfying $|g_i(x)| \leq B$. Such IFSs also produced attractor sets which strongly suggest convergence under the Hausdorff metric. Empirical evidence of this is shown in Fig. 7.1 for the functions $g_i = \cos(b_i x)$, $\sin(b_i x$, and $\tanh(b_i x)$, where each b_i was randomly selected from $[-4, -1] \cup [1, 4]$. For additional examples, see the appendix. Determining precisely under what conditions this occurs would be an interesting point of investigation.

Predicting IFS Parameters

There are many ways in which we could extend our neural network implementation to predict IFS parameters. Of course, different model configurations could always be tested, those with a higher complexity would likely produce the best results, should one have the memory for it. However, one could also attempt transfer learning. This is the idea of taking a model already trained to perform a given task, and retraining it to perform a similar task. Since the model trained on two-function IFSs had a higher accuracy than the other models, perhaps we could take this pre-trained two-function model and attempt to train it on four-function IFSs. The majority of the learning would then only occur in the last layer as there is a different number of outputs.

Since scaling the additive parameters was likely the cause of increased sparsity in the attractors of the model produced IFSs, one could train a model only on fractals that have not been scaled. This is feasible if one adopts a larger viewing region.

One could also add features to the loss function, such as including an extra component found from reconstructing the attractor of the model produced IFS. This additional value could then be calculated multiple ways. For example, one could calculate the pixel-wise difference between the input and the reconstructed attractor, or calculate the Hausdorff distance between the approximated attractor sets, amongst others. Though, implementing these methods would be extremely computationally expensive.

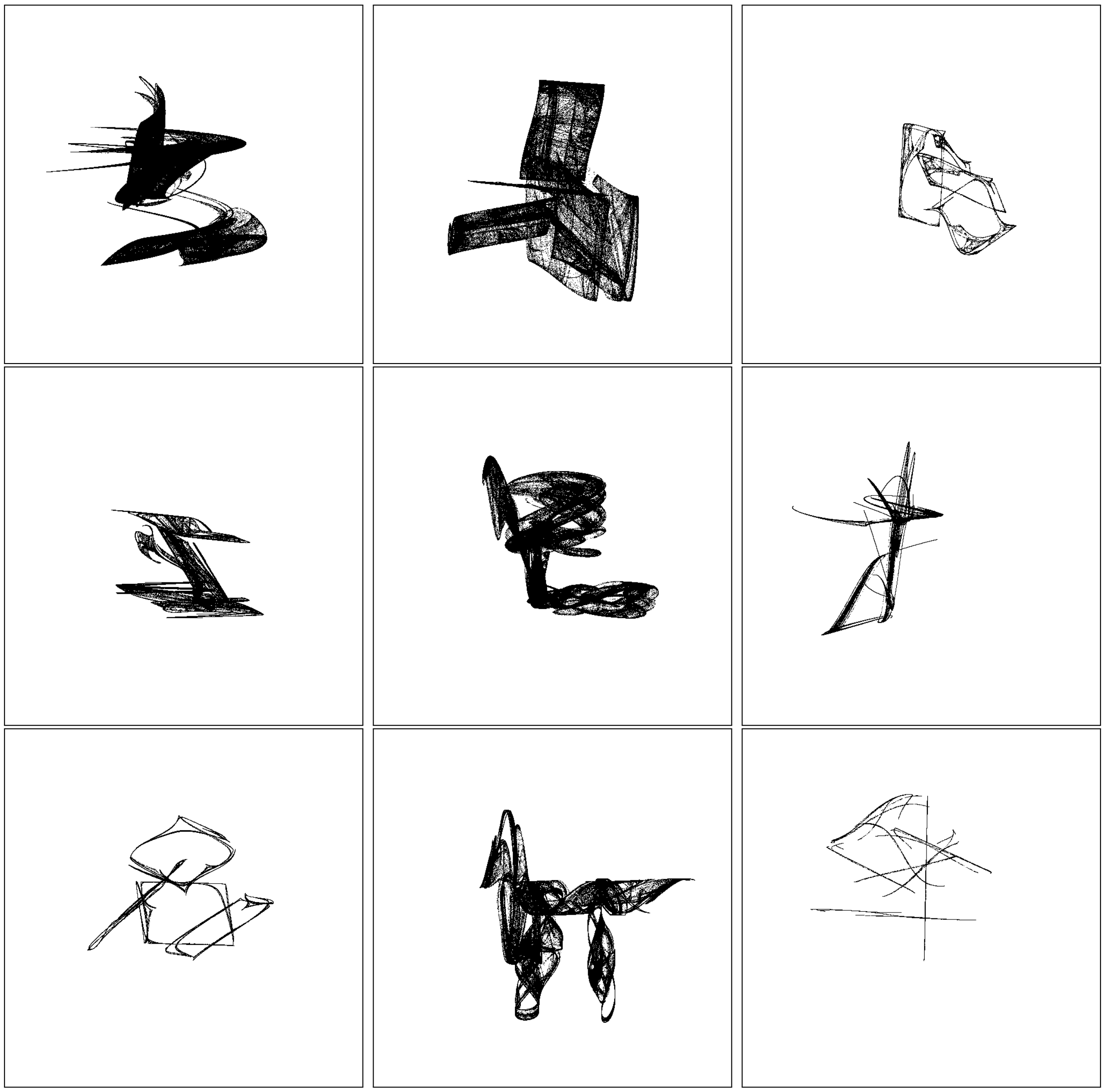


Figure 7.1: Examples of attractors created from bounded range functions. The row represents the g_1 function, and the column represents the g_2 function. The first row and column corresponds to cosine, the second to sine, and the third to the hyperbolic tangent function.

To increase the visual similarity between the input of the model with the attractor of the model produced IFS, one could also use the network output to generate an initial population of a swarm intelligence method, or genetic algorithm. This could be done by either perturbing the output, or using various saved points of the model throughout the learning process. Implementing this could then refine the values, as well as significantly reduce the time necessary for the swarm intelligence method, or genetic algorithm to converge.

Bibliography

[1] Shougeng, H., Cheng, Q., Wang, L., and Xie, S.: Multifractal characterization of urban residential land price in space and time. In: Applied Geography. **34**, pp. 161170 (2012). doi:10.1016/j.apgeog.2011.10.016.

[2] Hohlfeld, R. G., and Cohen, N.: Self-similarity and the geometric requirements for frequency independence in Antennae. In: Fractals. **7** (1), pp. 7984 (1999). doi:10.1142/S0218348X99000098.

[3] Vrscay, E.R.: Moment and collage methods for the inverse problem of fractal construction with iterated function systems. In: Peitgen, H.O., et al. (eds.) Fractals in the Fundamental and Applied Sciences. Elsevier, Amsterdam (1991)

[4] Abiko, T., Kawamata, M.: IFS coding of non-homogeneous fractal images using Gröbner basis. In: Proceedings of the IEEE International Conference on Image Processing, pp. 2529 (1999)

[5] Berkner, K.: A wavelet-based solution to the inverse problem for fractal interpolation functions. In: Lvy Vhel, J., Lutton, E., Tricot, C. (eds.) Fractals in Engineering, pp. 8192. Springer, London (1997) doi: 10.1007/978-1-4471-0995-2_7

[6] Nettleton, D.J., Garigliano, R.: Evolutionary algorithms and a fractal inverse problem. Biosystems 33, 221231 (1994). doi: 10.1016/0303-2647(94)90007-8

[7] Quirce J., Iglesias A., Glvez A.: Cuckoo search algorithm approach for the IFS Inverse Problem of 2D binary fractal images. In: Tan Y., Takagi H., Shi Y. (eds) Advances in Swarm Intelligence, pp. 543-551. Springer, Cham. (2017). doi: 10.1007/978-3-319-61824-1_59

[8] Goodfellow I., Bengio Y. and Courville A.: Deep learning. In: MIT Press (2016) Available via `http://www.deeplearningbook.org`

[9] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. V. D., Schrittwieser, J., et al.: Mastering the game of Go with deep neural networks and tree search. In: Nature **529** (7587), pp. 484-489 (2016)

[10] Barnsley, M.: Fractals Everywhere, 2nd edn. Morgan Kaufmann, San Francisco (1993)

[11] Hutchinson, J.: Fractals and self-similarity. In: Indiana University Mathematics Journal, (**30**) 713 - 747 (1981)

[12] Henrikson, J.: Completeness and total boundedness of the hausdorff metric. In: MIT Undergraduate Journal of Mathematics, pp. 69 - 80 (1999)

[13] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S.: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. In: Neural Networks. **6**, pp. 861-867 (1993)

[14] Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L.: The expressive power of neural networks: a view from the width. In: Advances in neural information processing systems, pp. 6231-6239 (2017)

[15] Meng, Q., Chen, W., Wang, Y., Ma, Z.-M., and Liu, T.-Y.: Convergence analysis of distributed stochastic gradient descent with shuffling. In: Neurocomputing. **337**, pp. 46-57 (2019)

[16] Kingma, D. P, and Ba, J., K.: Adam: A method for stochastic optimization. In: arXiv preprint arXiv:1412.6980, (2014)

[17] Srivastava N., Hinton G., Krizhevsky A., Sutskever I. and Salakhutdinov R.: Dropout: a simple way to prevent neural networks from overfitting. In: The journal of machine learning research. pp. 1929-58 (2014)

[18] Ioffe S. and Szegedy C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: arXiv preprint arXiv:1502.03167v3 (2015)

[19] Santurkar, S., Tsipras, D., Ilyas, A. and Madry, A.: How does batch normalization help optimization?. In: Advances in Neural Information Processing Systems. pp. 2483-2493 (2018)

[20] Krizhevsky, A., Sutskever, I., and Hinton, G. E.: ImageNet Classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097-1105 (2012)

[21] Simonyan, K., and Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In :arXiv preprint arXiv:1409.1556 (2014)

[22] Szegedy, C., Liu, W., Jia, Y., Pierre Sermanet, Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1-9 (2015)

[23] He K., Zhang X., Ren S. and Sun J.: Deep Residual Learning for Image Recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV. pp. 770-778 (2016). doi: 10.1109/CVPR.2016.90

[24] Xie S., Girshick R., Dollr P., Tu Z. and He K.: Aggregated Residual Transformations for Deep Neural Networks. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI. pp. 5987-5995 (2017). doi: 10.1109/CVPR.2017.634

[25] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A.: Inception-v4, inception-resnet and the impact of residual connections on learning. In: Thirty-first AAAI conference on artificial intelligence. (2017)

Appendix A

Supplementary Fractals

Here we provide some supplementary images of attractors to give a more well-rounded depiction of what those created from nonaffine functions can look like.

A.1 Bounded Derivative Function Attractors

Supplementary images of attractors made from IFSs consisting of functions with a derivative of $M = 1$ are given here. These can be found in Fig. A.1, Fig. A.2, and Fig. A.3.

A.2 Bounded Range Function Attractors

Additional examples of attractors corresponding to IFSs possessing range bounded functions are given in this section. These can be found in Fig. A.4, Fig. A.5, and Fig. A.6.

A.3 Piecewise Affine Function Attractors

More examples of randomly initialized, piecewise affine function consisting IFS attractors can be found in Fig. A.7. Each row of images in this figure has a different piecewise boundary.

A.4 Model Produced IFS Attractors

Additional examples of model produced IFS attractors that approximate the input image quite well for the two-function IFS neural network can be found in Fig. A.8.

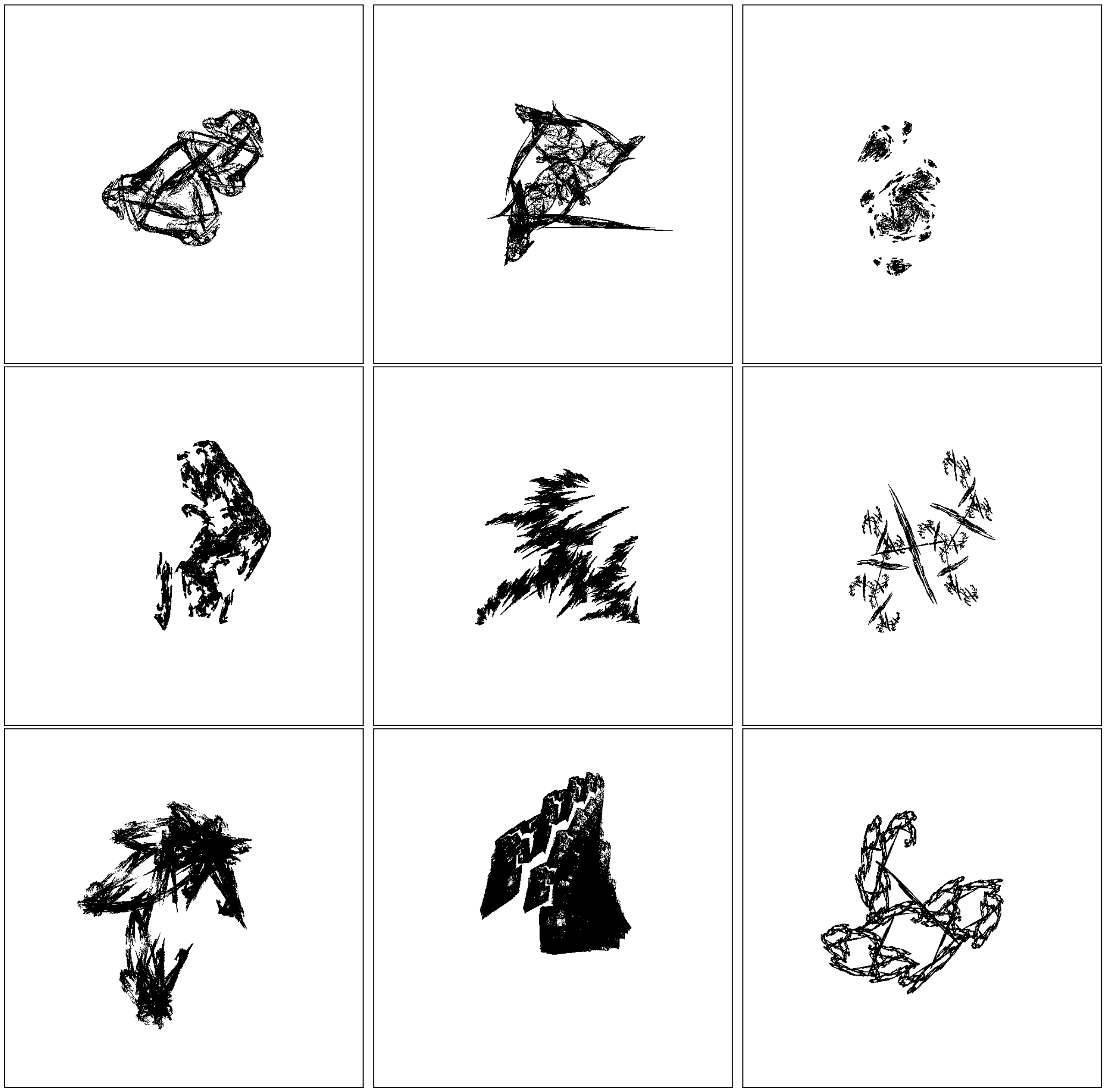


Figure A.1: Supplementary images of attractors pertaining to IFSs consisting of derivative bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = cos(b_i x)b_i$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)/b_i$, $g_2 = sin(b_i x)/b_i$, and $g_2 = tanh(b_i x)/b_i$, respectively.

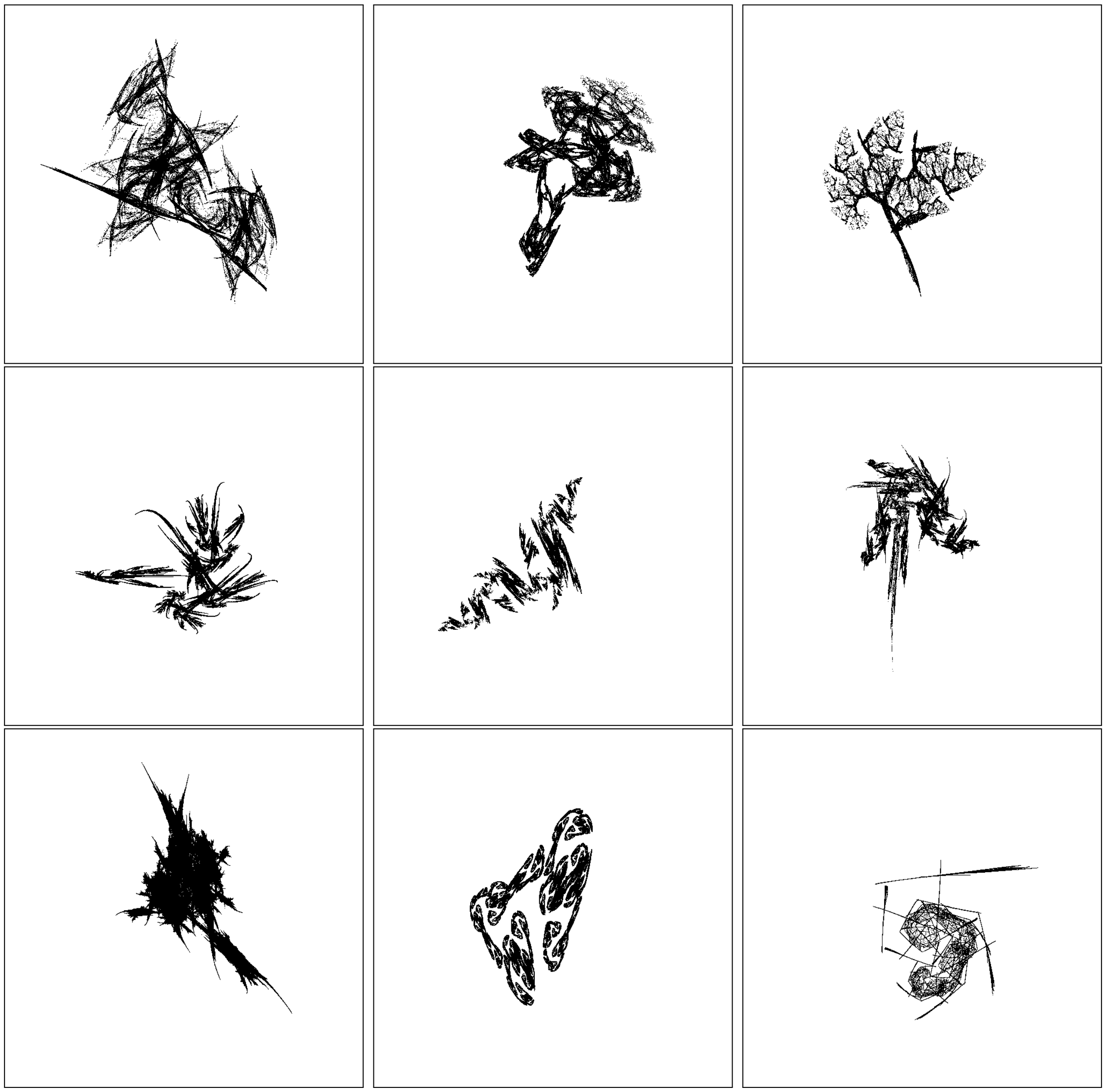


Figure A.2: Supplementary images of attractors pertaining to IFSs consisting of derivative bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = cos(b_i x)b_i$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)/b_i$, $g_2 = sin(b_i x)/b_i$, and $g_2 = tanh(b_i x)/b_i$, respectively.

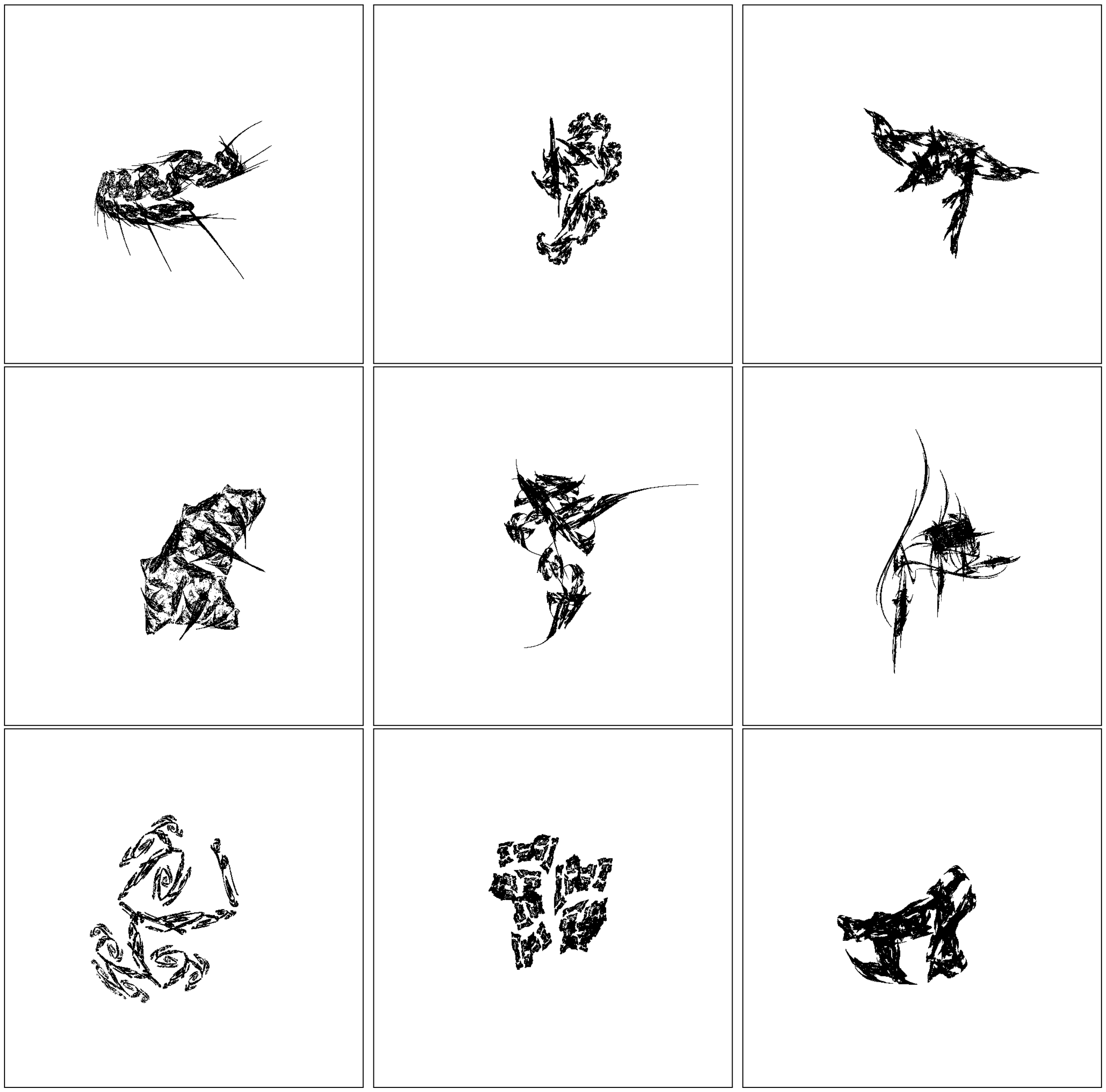


Figure A.3: Supplementary images of attractors pertaining to IFSs consisting of derivative bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = cos(b_i x) b_i$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)/b_i$, $g_2 = sin(b_i x)/b_i$, and $g_2 = tanh(b_i x)/b_i$, respectively.

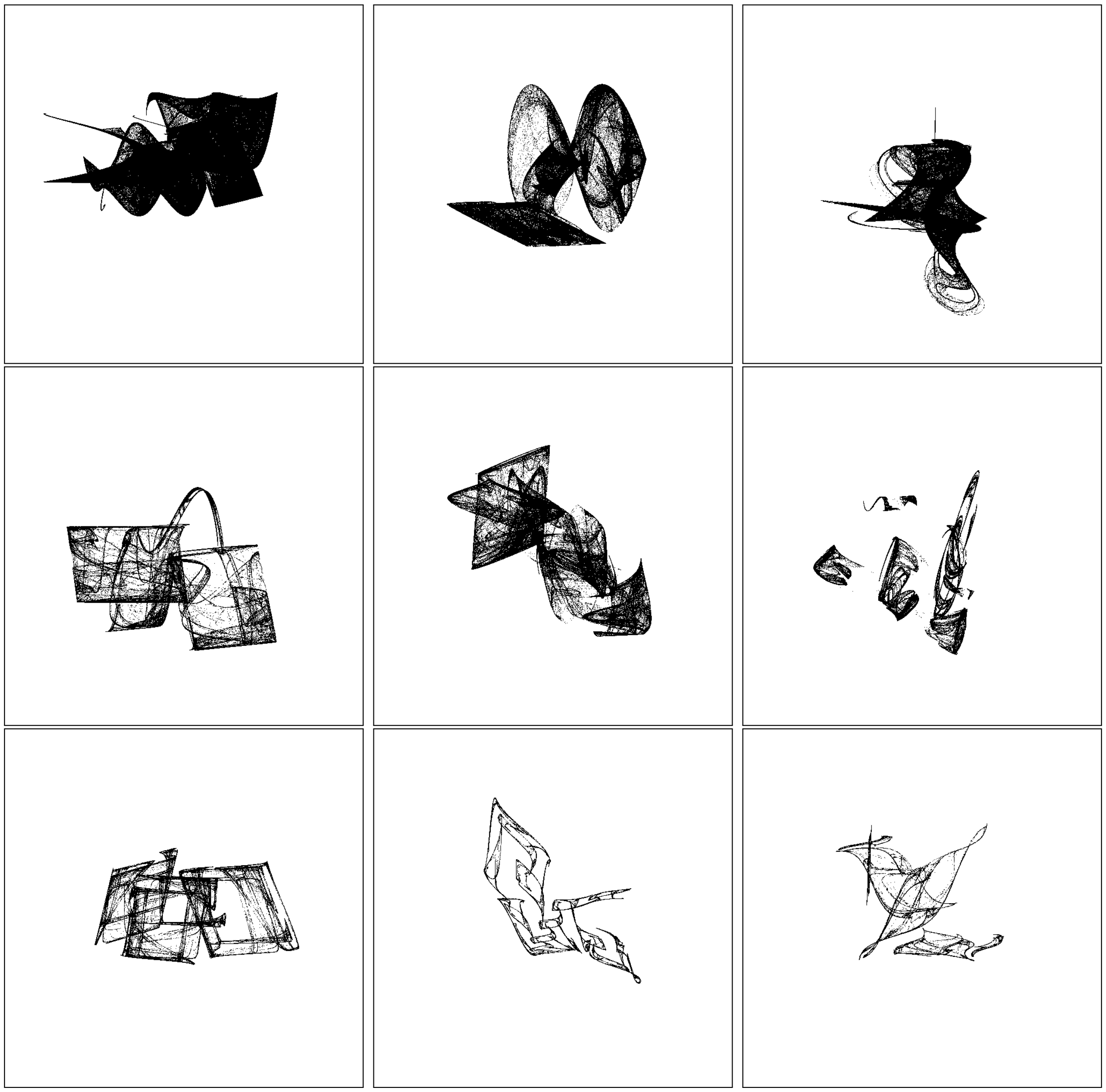


Figure A.4: Supplementary images of attractors pertaining to IFSs consisting of range bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = cos(b_i x)$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)$, $g_2 = sin(b_i x)$, and $g_2 = tanh(b_i x)$, respectively.

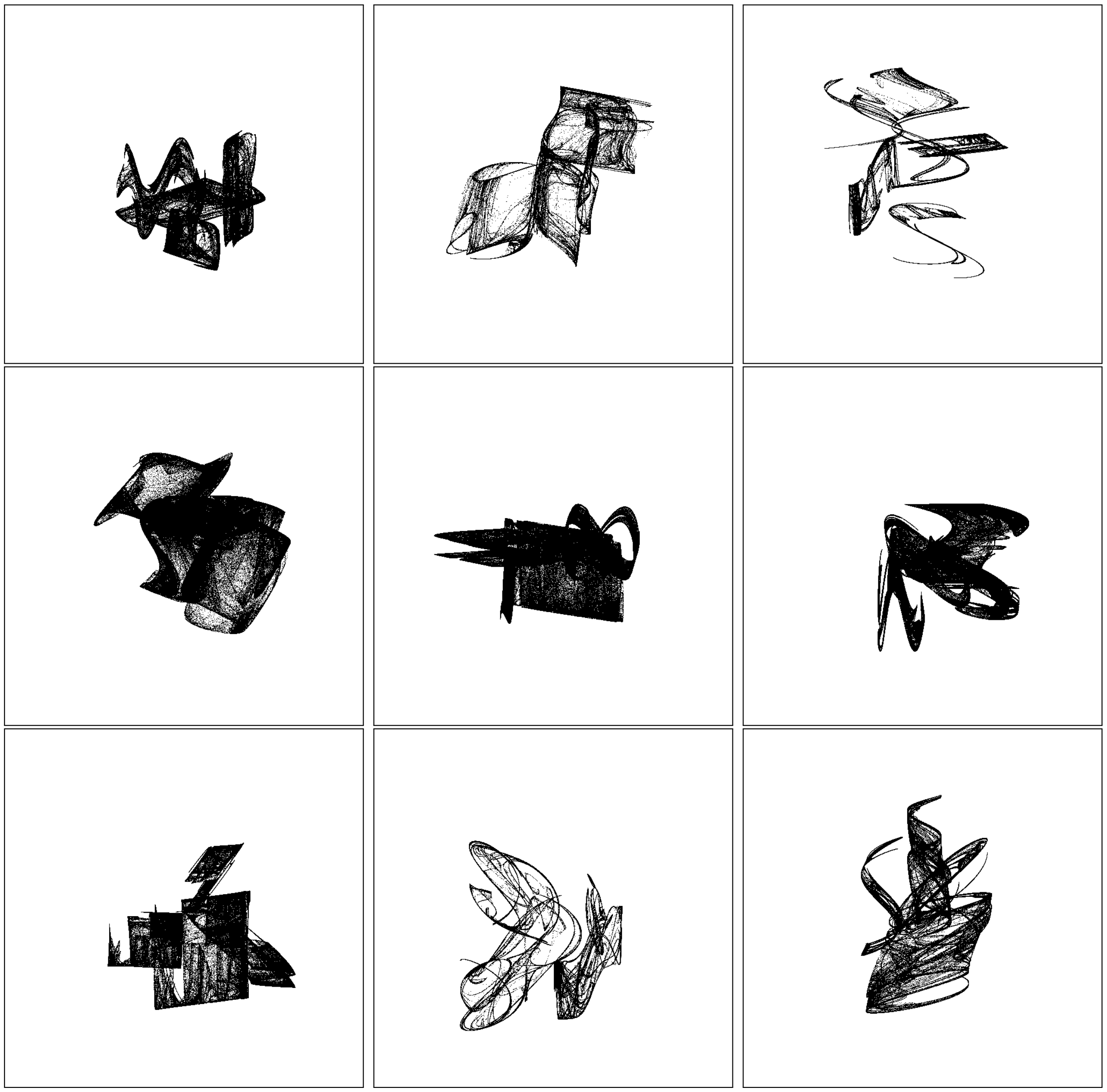


Figure A.5: Supplementary images of attractors pertaining to IFSs consisting of range bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = sin(b_i x)$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)$, $g_2 = sin(b_i x)$, and $g_2 = tanh(b_i x)$, respectively.

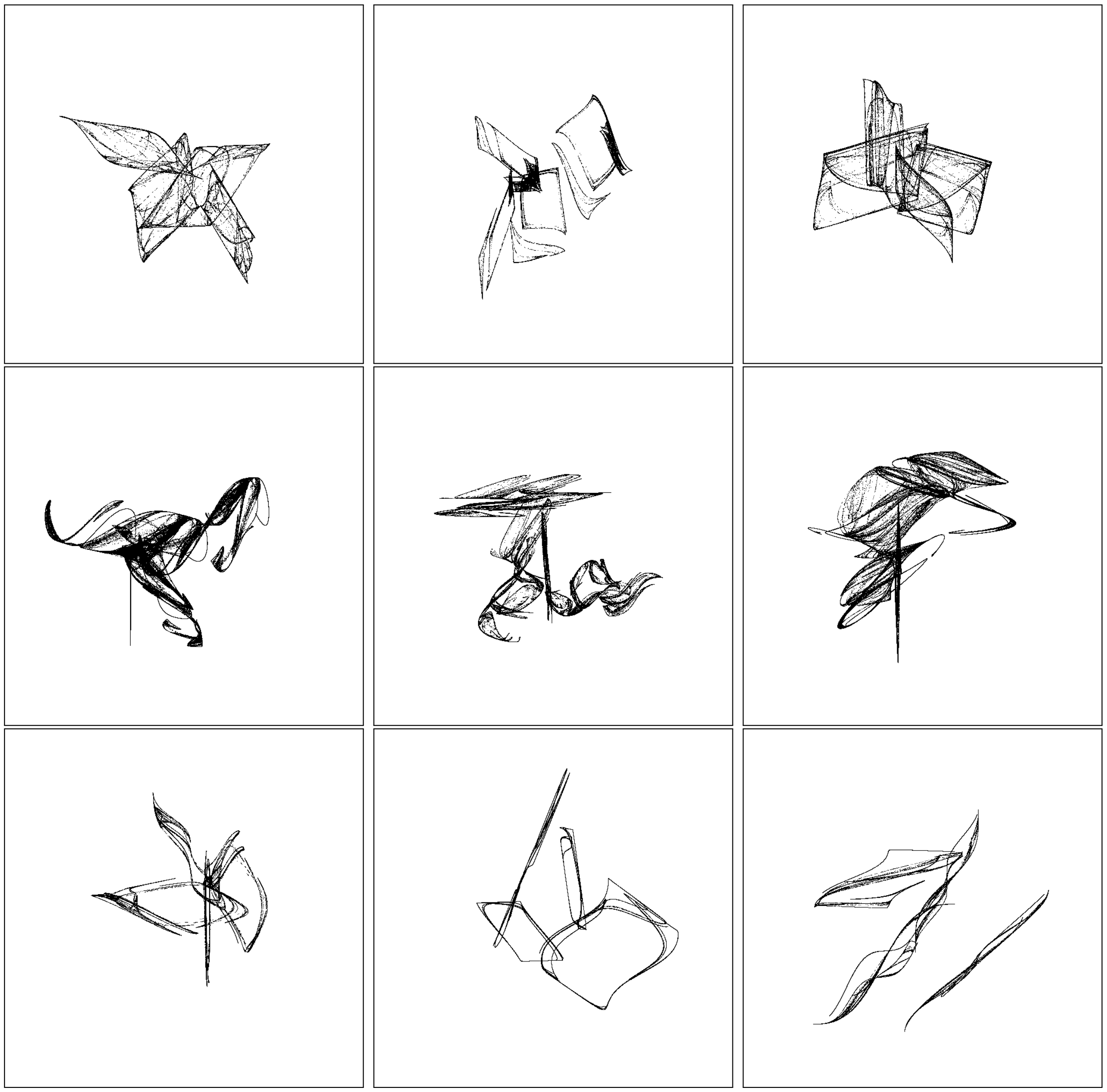


Figure A.6: Supplementary images of attractors pertaining to IFSs consisting of range bounded functions. Each of the IFSs in this figure have $g_1 = g_3 = tanh(b_i x)$. Additionally, each row going from top to bottom has $g_2 = g_4$ such that $g_2 = cos(b_i x)$, $g_2 = sin(b_i x)$, and $g_2 = tanh(b_i x)$, respectively.

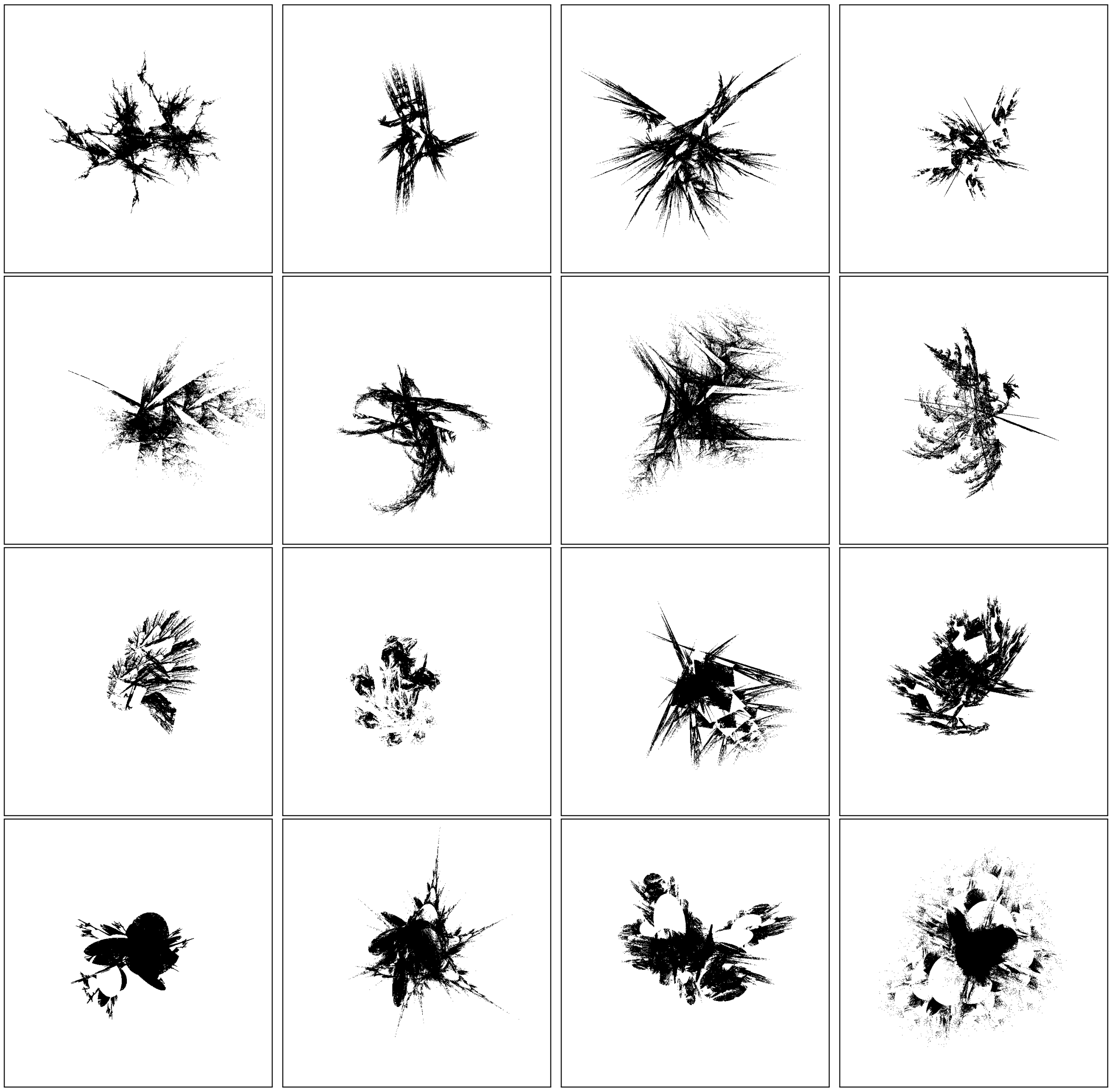


Figure A.7: Supplementary images of attractors pertaining to IFSs consisting of piecewise functions. Each row has the piecewise boundary defined by $x = 0$ when $y > 0$ and $y = 0$ when $x > 0$, $x + y = 0$, $|x| + |y| = 1$, and $x^2 + y^2 = 1$, going from top to bottom, respectively.

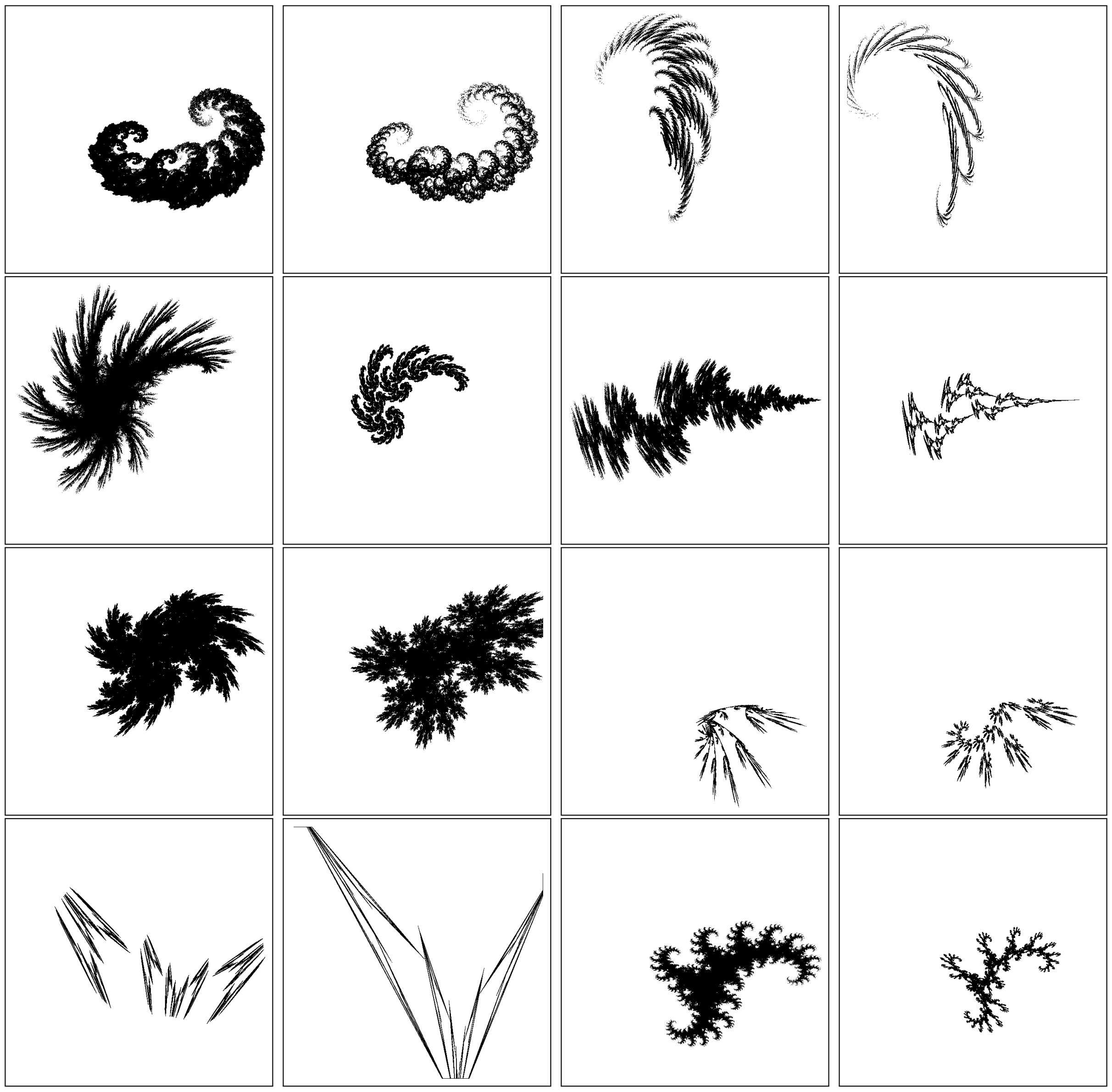


Figure A.8: Supplementary images of model produced IFS attractors compared to their input for the two-function IFS neural network. The first and third column are the input images to the attractor to the right, respectively.

Appendix B

Source Code

B.1 Generating Fractal Databases

In this section we present some code that can be compiled and used to generate databases of fractals. Note that this is a subset of the original code; many additional features have been removed in order to shorten the length of the files. Additionally, there are some sections pertaining to concepts that were not discussed. This code was written in the C programming language and requires the libpng library.

```
/*Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: Fractals.h
 */
#define HEIGHT 640
#define WIDTH 640

struct Fractal{
        double dimension, stddevx, stddevy, *xs, *ys, **genome;
        int fracnum, numfuncs, numpoints, numb, dist, avgx, avgy, **bm, *colours, coloured;
};

double f(double *val, double point, double functype);
int funcind(int funcnum, double **genome);
int funcaddind(int funcnum, double **genome);
int multindjump(int functype);
int addindjump(int functype);
int piecewisecond(double x, double y);
void func(double *x, double *y, double **genome, int funcnum, double functype);
double validranddouble(double functype);
void generatemults(double **genome, double functype, int *multparams);
void generateadds(double **genome, double functype, int *addparams);
void generategenome(struct Fractal *frac, int *restrictions, int numrestrictions, int
    disperse);
void ordergenome(int numfuncs, double **genome);
double funcdeterminant(double a, double b, double c, double d);
int validatefunc(double a, double b, double c, double d);
double ** mallocgenome(int numfuncs);
void initializefrac(struct Fractal *frac, int numfuncs, int numpoints);
double generatepoints(struct Fractal *frac);
int generatefrac(struct Fractal *frac);
int * pointtocoord(double x, double y, double minx, double maxx, double miny, double maxy);
void generatematrix(struct Fractal *frac, double *window);
void freegenome(struct Fractal *frac);
void freefrac(struct Fractal *frac);
int lenfile(char *filename);

/*Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME:  Fractals.c
 *
 * This file contains functions that are used to
 * generate fractals, both affine and non-affine
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include "Fractals.h"
#include "vecio.h"
#include "matvec_read.h"
#define DOTSIZE 1 //must be an odd positive integer

double f(double *val, double point, double functype){
    /* This function is used to compute non-affine transformations
     * on a single coordinate point. (ie. only x for the point (x, y))
     * where the transformation type is represented as an integer,
     * functype, computed with a randomly generated value, val,
     * stored in the IFS genome
     */
    double newval;
    if (functype == 1){
        newval = val[0]*tanh(val[1]*point);
    }
    if (functype == 2){
        newval = val[0]*sin(val[1]*point);
    }
    if (functype == 3){
        newval = val[0]*tanh(val[1]*point);
    }
    return newval;
}

int funcind(int funcnum, double **genome){
    /* This function returns the starting index
     * of funcnumth (like nth) function in the
     * fractals genome. This function is calculates
     * the starting index of the multiplicative
     * parameters
     */
    int i;
    int ind = 0;
    int j;
    for (i = 0; i < funcnum; i++){
        j = genome[3][i];
        if (j == 0) ind += 4;
        else if (j<11) ind += 8;
    }
    return ind;
}

int funcaddind(int funcnum, double **genome){
    /* This function returns the starting index
     * of funcnumth (like nth) function in the
     * fractals genome. This function is calculates
     * the starting index of the additive
     * parameters
     */
    int i;
    int ind = 0;
    int j;
    for (i = 0; i < funcnum; i++){
        j = genome[3][i];
        if (j < 10) ind += 2;
        else ind += 4;
    }
    return ind;
}

int multindjump(int functype){
    /* This function is similar to funcind */
    if (functype == 0) return 4;
    else if (functype < 11) return 8;
    else return 0;
}

int addindjump(int functype){
    /* This function is similar to funcindadd */
    if (functype < 10) return 2;
    else if (functype == 10) return 4;
    else return 0;
}
```

```c
int piecewisecond(double x, double y){
    /* This function is used for piecewise affine IFSs
     * It calculates whether a point is on one side of
     * the piecewise boundary,  or the other, returning
     * either 0 or 1
     * With the current if statement the piecewise
     * boundary is the circle of radius 1/2 centered at 0
     */
    if (fabs(x) + fabs(y) < 0.5) return 1;
    else return 0;

}

void func(double *x, double *y, double **genome, int funcnum, double functype){
    /* This function computes the transformation of a point (x, y) by a
     * function in an IFS. The specific function in the IFS is given by
     * an integer, funcnum, then the parameters of that function are obtained
     * from the IFS genome. The function type is then used to compute the
     * transformation.
     *
     * if functype is 0:    affine transformation   new_x = ax + by + c
     *                1:    2 to 1 trig mapping     new_x = acos(bx) + ccos(dy) + e
     *                2:    2 to 1 trig mapping     new_x = acos(by) + csin(dy) + e
     *                3:    2 to 1 trig mapping     new_x = acos(by) + ctanh(dy) + e
     *                4:    2 to 1 trig mapping     new_x = asin(bx) + ccos(dy) + e
     *                5:    2 to 1 trig mapping     new_x = asin(bx) + csin(dy) + e
     *                6:    2 to 1 trig mapping     new_x = asin(bx) + ctanh(dy) + e
     *                7:    2 to 1 trig mapping     new_x = atanh(bx) + ccos(dy) + e
     *                8:    2 to 1 trig mapping     new_x = atanh(bx) + csin(dy) + e
     *                9:    2 to 1 trig mapping     new_x = atanh(bx) + ctanh(dy) + e
     *                10:   piecewise affine        new_x = {ax+by + c, x < 0
     *                                                      dx+ey + f, x >= 0
     *
     */
    double oldx = *x;
    double oldy = *y;
    int ind = funcind(funcnum, genome);
    int addind = funcaddind(funcnum, genome);
    if (functype == 0){
        (*x) = genome[0][ind]   * oldx + genome[0][ind+1] * oldy + genome[1][addind];
        (*y) = genome[0][ind+2] * oldx + genome[0][ind+3] * oldy + genome[1][addind+1];
    }
    else if (functype == 1){
        (*x) = f(&(genome[0][ind]), oldx, 1) + f(&(genome[0][ind+2]), oldy, 1) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 1) + f(&(genome[0][ind+6]), oldy, 1) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 2){
        (*x) = f(&(genome[0][ind]), oldx, 1) + f(&(genome[0][ind+2]), oldy, 2) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 1) + f(&(genome[0][ind+6]), oldy, 2) + genome
```

```
            [1][funcnum*2+1];
    }
    else if (functype == 3){
        (*x) = f(&(genome[0][ind]), oldx, 1) + f(&(genome[0][ind+2]), oldy, 3) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 1) + f(&(genome[0][ind+6]), oldy, 3) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 4){
        (*x) = f(&(genome[0][ind]), oldx, 2) + f(&(genome[0][ind+2]), oldy, 1) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 2) + f(&(genome[0][ind+6]), oldy, 1) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 5){
        (*x) = f(&(genome[0][ind]), oldx, 2) + f(&(genome[0][ind+2]), oldy, 2) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 2) + f(&(genome[0][ind+6]), oldy, 2) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 6){
        (*x) = f(&(genome[0][ind]), oldx, 2) + f(&(genome[0][ind+2]), oldy, 3) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 2) + f(&(genome[0][ind+6]), oldy, 3) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 7){
        (*x) = f(&(genome[0][ind]), oldx, 3) + f(&(genome[0][ind+2]), oldy, 1) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 3) + f(&(genome[0][ind+6]), oldy, 1) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 8){
        (*x) = f(&(genome[0][ind]), oldx, 3) + f(&(genome[0][ind+2]), oldy, 2) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 3) + f(&(genome[0][ind+6]), oldy, 2) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 9){
        (*x) = f(&(genome[0][ind]), oldx, 3) + f(&(genome[0][ind+2]), oldy, 3) + genome[1][
            funcnum*2];
        (*y) = f(&(genome[0][ind+4]), oldx, 3) + f(&(genome[0][ind+6]), oldy, 3) + genome
            [1][funcnum*2+1];
    }
    else if (functype == 10){
        if (piecewisecond(oldx, oldy)) {
            (*x) = genome[0][ind]   * oldx + genome[0][ind+1] * oldy + genome[1][addind];
            (*y) = genome[0][ind+2] * oldx + genome[0][ind+3] * oldy + genome[1][addind+1];
        }
        else {
            (*x) = genome[0][ind+4] * oldx + genome[0][ind+5] * oldy + genome[1][addind+2];
            (*y) = genome[0][ind+6] * oldx + genome[0][ind+7] * oldy + genome[1][addind+3];
        }
    }
}

double validranddouble(double functype){
    /* This function generates random values
     * within a specific range for IFS parameters
     */
    double min;
    double range;
    // for most parameters:
    if (functype >= 0) {
        min = -1.;
        range = 2.;
    }
    //for the b values in bounded derivative functions
    else if (functype == -1){
        min = 1.;
        range = 4.;
    }
    double val = (double)rand()/RAND_MAX*range + min;
    if (functype == -1){
        min = (double)rand()/RAND_MAX;
        if (min < 0.5) val *= -1.;
    }
    return val;
}

void generatemults(double **genome, double functype, int *multparams){
    /* This function generates the multiplicative parameters
     * of each function in an IFS. ie., the parameters that are not
     * the +c or +e in the functions defined in the func() function
     *
     * The parameters are generated using validranddouble function
     * and the function is checked if it satisfies contractivity
     * conditions. If the function does not satisfy the contractivity
     * conditions, all parameters for that function are regenerated.
     */
    int numparams, i;
    int pass = 1;
    if (functype == 0) numparams = 4;
    else  numparams = 8;

    i = *multparams;
    if (functype == 0){
        while (pass != 0){
            for (int j = i; j < i + 4; j++){
                genome[0][j] = validranddouble(functype);
            }
            pass = validatefunc(genome[0][i], genome[0][i+1], genome[0][i+2], genome[0][i
                +3]);
        }
```

```
            pass = 1;
        }
        else if (functype < 10){
            while (pass != 0){
                for (int j = 0; j < 4; j++){
                    genome[0][i + 2*j] = validranddouble(functype);
                }
                pass = validatefunc(genome[0][i], genome[0][i+2], genome[0][i+4], genome[0][i
                    +6]);
            }
            for (int j = 0; j < 4; j++){
                genome[0][i + 2*j+1] = validranddouble(-1);
            }
            pass = 1;
        }
        else if (functype == 10){
            while (pass != 0){
                for (int j = i; j < i + 4; j++){
                    genome[0][j] = validranddouble(functype);
                }
                pass = validatefunc(genome[0][i], genome[0][i+1], genome[0][i+2], genome[0][i
                    +3]);
            }
            pass = 1;
            while (pass != 0){
                for (int j = i+4; j < i + 8; j++){
                    genome[0][j] = validranddouble(functype);
                }
                pass = validatefunc(genome[0][i+4], genome[0][i+5], genome[0][i+6], genome[0][i
                    +7]);
            }
            pass = 1;
        }
        *multparams += numparams;
        return;
}

void generateadds(double **genome, double functype, int *addparams){
    /* This function generates the additive parameters
     * for each function in the IFS. ie., the +c or +e in the
     * functions defined in the func() function.
     *
     * Since these parameters do not affect contractivity,
     * they are generated to be a random number between
     * -1 and 1.
     */
    int i;
    int numparams;
    if (functype < 10) numparams = 2;
    else numparams = 4;

    for (i = *addparams; i < *addparams + numparams; i++){
        genome[1][i] = (double)rand()/RAND_MAX*2. - 1.;
    }
    *addparams += numparams;
    return;
}

void generategenome(struct Fractal *frac, int *restrictions, int numrestrictions, int
    disperse){
    /* This function generates the genome of a fractal, ie., the set of vectors that
     * that contains all the parameters used to generate the fractal.
     * NOTE: This function assumes that initializefrac has been called
     *
     * INPUTS:
     *      frac            - the fractal struct for which the genome is being
     *                        generated and stored in
     *      restrictions    - a vector containing function types for which the
     *                        fractal can't have based on the function types
     *                        defined in the func() function. Eg., if restrictions
     *                        were the vector {2,3,4} then the fractal can have
     *                        functypes {0,1,5,6,7,8,9,10}
     *      numrestrictions - an integer corresponding to the length of the
     *                        restrictions vector.
     *      disperse        - an integer defining the dispersion of the functions
     *                        according to the restrictions.
     *      if disperse is 0: there are no restrictions, function types are randomly
     *                        generated so long as they're not in the restrictions
     *                        vector. There is no guaruntee that a specific function
     *                        type will be present in the resulting fractal.
     *                     1: this option requires at least 2 different types of
     *                        functions. However, if there are more than 2 allowed
     *                        types of functions then it is not guarunteed to contain
     *                        each type.
     *                     2: this option will enforce that at least one of each
     *                        function type will be in the resulting fractal.
     */
    int i, j;
    int multparams = 0;
    int addparams = 0;
    int pass = 1;
    int premade = 0;
    int numfunctypes = 11;
    double **genome = frac -> genome;
    if (disperse == 1) {
        premade = 2;
        for (i = 0; i < 2; i++){
            while (pass != 0){
                pass = 0;
                genome[3][i] = rand()%numfunctypes;
                for (j = 0; j < numrestrictions; j++){
                    if (genome[3][i] == restrictions[j]) pass = 1;
                    if ((i == 1) && (genome[3][i] == genome[3][i-1])) pass = 1;
                }
            }
        }
    }
    if (disperse == 2) {
        for (i = 0; i < numfunctypes; i++){
            pass = 0;
            for (j = 0; j < numrestrictions; j++){
                if (i == restrictions[j]) pass = 1;
            }
            if (pass == 0) {
                genome[3][premade] = i;
                premade++;
            }
        }
        pass = 1;
    }
    for (i = premade; i < frac -> numfuncs; i++){
        while (pass != 0){
            pass = 0;
            genome[3][i] = rand()%numfunctypes;
            for (j = 0; j < numrestrictions; j++){
                if (genome[3][i] == restrictions[j]) pass = 1;
            }
        }
        pass = 1;
    }
    dsortvec(frac -> numfuncs, genome[3]);
    for (i = 0; i < frac -> numfuncs; i++){
        generatemults(genome, genome[3][i], &multparams);
        generateadds(genome, genome[3][i], &addparams);
        genome[2][i] = 1./(double)frac -> numfuncs;
    }
    return;
}

void ordergenome(int numfuncs, double **genome){
    /* This function is used to order the functions in the
    * fractal genome. This is done by sorting rows of a
    * matrix where each row represents a function
    */
    int i,j, curind, indf;
    double **genomefuncs = dmatmem(numfuncs, 22);
    /* Setting values in the genomefuncs matrix */
    for (i = 0; i < numfuncs; i++){
        genomefuncs[i][0] = genome[3][i]; //place functype first
        curind = 1;
        indf = funcind(i, genome);
        for (j = 0; j < multindjump(genome[3][i]); j++){
            genomefuncs[i][curind] = genome[0][indf + j];
            curind++;
        }
        indf = funcaddind(i, genome);
        for (j = 0; j < addindjump(genome[3][i]); j++){
            genomefuncs[i][curind] = genome[1][indf + j];
            curind++;
        }
        genomefuncs[i][curind] = genome[2][i];
    }
    dsortmatrows(numfuncs, 4, 1, genomefuncs);

    /* Converting back to genome of the fractal */
    for (i = 0; i < numfuncs; i++){
        genome[3][i] = genomefuncs[i][0];
    }
    for (i = 0; i < numfuncs; i++){
        curind = 1;
        indf = funcind(i, genome);
        for (j = 0; j < multindjump(genomefuncs[i][0]); j++){
            genome[0][indf + j] = genomefuncs[i][curind];
            curind++;
        }
        indf = funcaddind(i, genome);
        for (j = 0; j < addindjump(genomefuncs[i][0]); j++){
            genome[1][indf + j] = genomefuncs[i][curind];
            curind++;
        }
        genome[2][i] = genomefuncs[i][curind];
    }
    return;
}

double funcdeterminant(double a, double b, double c, double d){
    /* This function is used to calculate the determinant
     * of an affine transformation matrix in order to
     * check contractivity.
     */
    return a*d - c*b;
}

int validatefunc(double a, double b, double c, double d){
    /* This function checks if an affine transformation
     * satisfies contractivity conditions. This function
     * returns 0 if contractive, 1 otherwise.
     *
     * if an affine transformation is given by:
     *
     *        [x]    [a b][x]    [e]
     *        [y] = [c d][y] + [f]
     *
     * then these contractivity conditions are
     *  a^2 + c^2 < 1
     *  b^2 + d^2 < 1
     *  a^2 + b^2 + c^2 + d^2 - det(A)^2 < 1
     */
    double val1, val2;
    double det = funcdeterminant(a,b,c,d);
    if ((val1 = a*a + c*c)>=1) return 1;
    else if ((val2 = b*b + d*d)>=1) return 1;
    else if ((val1 + val2 - det*det) >= 1) return 1;
    else return 0;
}

double ** mallocgenome(int numfuncs){
    /* This function allocates memory for the genome of an IFS
     * The genome consists of 4 vectors
     * genome[0] is the vector of multiplicative parameters
     * genome[1] is the vector of additive parameters
     * genome[2] is the vector of probabilities for each function
     * genome[3] is the vector of functypes
     */
    double **genome;
    if ((genome = (double **)malloc(4*sizeof(double *))) == NULL){
        fprintf(stderr, "Malloc failed (generate genome)\n");
        exit(1);
    }
    if (((genome[0] = (double *)malloc(8*numfuncs*sizeof(double))) == NULL)||
        ((genome[1] = (double *)malloc(4*numfuncs*sizeof(double))) == NULL)||
        ((genome[2] = (double *)malloc(numfuncs*sizeof(double))) == NULL)){
            fprintf(stderr, "Malloc failed (initializefrac)\n");
            exit(1);
    }
    if ((genome[3] = (double *)malloc(numfuncs*sizeof(double))) == NULL){
        fprintf(stderr, "Malloc failed (initializefrac 3)\n");
        exit(1);
    }
    return genome;
}

void initializefrac(struct Fractal *frac, int numfuncs, int numpoints){
    /* This function initializes a fractal structure. The number of
     * points and number of functions has to be defined before it
     * can be called. This function then allocates memory for the
     * x and y points that will be generated as well as allocating
     * memory for the matrix representing the picture of the fractal.
     *
     * All values corresponding to the fractal other than numfuncs,
     * numpoints, and whether the fractal is colours or not
     * are initialized to -1
     */
    int i;
    frac -> fracnum   = -1;
    frac -> numfuncs  = numfuncs;
    frac -> numpoints = numpoints;
    frac -> numb      = -1;
    frac -> avgx      = -1;
    frac -> avgy      = -1;
    frac -> stddevx   = -1;
    frac -> stddevy   = -1;
    frac -> dimension = -1;
    frac -> dist      = -1;
    frac -> coloured  = 1; //dont colour fractals by function by default
                           //to make them coloured by function by default
                           //change this to 0

    /* initialize genome */
    double **genome = mallocgenome(numfuncs);
    frac -> genome = genome;

    /* initialize xs, ys, and colour vector*/
    if ((frac -> xs = (double *)malloc(numpoints * sizeof(double))) == NULL){
        fprintf(stderr, "Malloc␣Failed.␣(initialize␣points)\n");
        exit(1);
    }
    if ((frac -> ys = (double *)malloc(numpoints * sizeof(double))) == NULL){
        fprintf(stderr, "Malloc␣Failed.␣(initialize␣points)\n");
        exit(1);
    }
    if ((frac -> colours = (int *)malloc(numpoints * sizeof(int))) == NULL){
        fprintf(stderr, "Malloc␣Failed.␣(initialize␣points)\n");
        exit(1);
    }
    /* initizlize the pixel map */
    int **bm;
    if ((bm = (int **)malloc(HEIGHT*sizeof(int *))) == NULL){
        fprintf(stdout, "Malloc␣Failed.␣(makematrix)\n");
        exit(1);
    }
    for (i = 0; i < HEIGHT; i++){
        if ((bm[i] = (int *)malloc(WIDTH*sizeof(int))) == NULL){
            fprintf(stderr, "Malloc␣Failed.␣(makematrix)\n");
            exit(1);
        }
    }
    frac -> bm = bm;
    return;
}

double generatepoints(struct Fractal *frac){
    /* This function generates the points corresponding
     * to a fractal. That is, it randomly picks a function
     * in the fractal and uses it to transform a random
     * point. A new function is then picked and transforms
     * the output from the last point. This continues until
     * numpoints points are generated. Additionally, the
     * first 100 points are thrown away to ensure that all
     * (or close to all) points correspond to the fractal.
     */
    int i,j;
    int funcnum;
    double p, num;
    double max = 0;
    double x = (double)rand()/RAND_MAX;
    double y = (double)rand()/RAND_MAX;
    double maxx = 0;
    double maxy = 0;
    for (i = 0; i < 100; i++){
        funcnum = rand()%frac -> numfuncs;
        func(&x, &y, frac -> genome, funcnum, frac -> genome[3][funcnum]);
    }
    for (i = 0; i < frac -> numpoints; i++){
        num = (double)rand()/RAND_MAX;
        p = 0.0;
        funcnum = 0;
        for (j = 0; j < frac -> numfuncs; j++){
            p += frac -> genome[2][j];
            if (num < p) {
                func(&x,&y,frac -> genome, funcnum, frac -> genome[3][j]);
                frac -> xs[i] = x;
                frac -> ys[i] = y;

                //note: colours get put to pixels in generatebm (below)
                //      and colours chosen are in PNGio.c
                frac -> colours[i] = funcnum;
                if (fabs(x) > max) max = fabs(x);
                if (fabs(y) > max) max = fabs(y);
                if (fabs(x) > maxx) maxx = fabs(x);
                if (fabs(y) > maxy) maxy = fabs(y);
                break;
            }
            else funcnum ++;
        }
    }
    return max;
}

int generatefrac(struct Fractal *frac){
    /* This function calls the generate points function.
     * The commented section is used to resized the affine
     * fractals if they were too large to fit into the -1 to 1
     * square, however, this is no longer used as non-affine
     * IFSs are not as simple to resize
     */
    double max = generatepoints(frac);
    int resized = 0;
    /*
    int i = 0;
    if (max > 1.0){
        for (int i = 0; i < 2 * frac -> numfuncs; i++){
            frac -> genome[1][i] /= max;
        }
        max = generatepoints(frac);
        resized = 1;
        i++;
    }
    */
    max += 1; // so the compiler doesn't give a warning for not using max
    return resized;
}

int * pointtocoord(double x, double y, double minx, double maxx, double miny, double maxy){
    /* This function is used to convert a point (x,y) to pixel coordinates
     * on a screen with viewing region [minx,maxx]x[miny,maxy]
     */
    int *coords = ivecmem(2);
    coords[0] = (int)(WIDTH/2  + WIDTH/2  * ((x - minx)/(maxx - minx)*2 - 1));
    coords[1] = (int)(HEIGHT/2 - HEIGHT/2 * ((y - miny)/(maxy - miny)*2 - 1));
    return coords;
}

void generatematrix(struct Fractal *frac, double *window){
    /* This function is used to transform the points of a fractal
     * to a matrix of size HEIGHT x WIDTH which will be used to
     * generate an image of the fractal.
     */
    int i,j,k,x,y;
    int dotsize = DOTSIZE; //positive odd integer - defines the size of a point
    int white = 255;
    int **bm = frac -> bm;

    //start off with a fully white image
    for (i = 0; i < HEIGHT; i++){
        for (j = 0; j < WIDTH; j++){
            bm[i][j] = white;
        }
    }
    //numb is the number of pixels corresponding to the attractor
    //avgx and avgy are the pixel centroid coordinates
    int numb = 0;
    int avgx = 0;
    int avgy = 0;
    int *coords;
    for (i = 0; i < frac -> numpoints; i++){
   coords = pointtocoord(frac ->xs[i], frac->ys[i], window[0],
                              window[1], window[2], window[3]);
        x = coords[0];
        y = coords[1];
        if (dotsize %2 != 0) {
             for (j = -1 * (dotsize -1)/2; j <= (dotsize - 1)/2; j++){
                 for (k = -1 * (dotsize -1)/2; k <= (dotsize -1)/2; k++){
                     if (x >= WIDTH  - dotsize/2 - 1) x = WIDTH  - dotsize/2 - 1;
                     if (y >= HEIGHT - dotsize/2 - 1) y = HEIGHT - dotsize/2 - 1;
                     if (x <= dotsize/2) x = dotsize;
```

```
                    if (y <= dotsize/2) y = dotsize;
                    if (bm[y+j][x+k] == 255){
                        avgx += x+k;
                        avgy += y+j;
                        numb += 1;
                    }
                    bm[y+j][x+k] = frac -> colours[i];
                }
            }
        }
    }
    frac -> avgx = avgx/(int)numb;
    frac -> avgy = avgy/(int)numb;
    frac -> numb = numb;
    return;
}

void freegenome(struct Fractal *frac){
    /* This function frees the genome memory */
    free(frac -> genome[0]);
    free(frac -> genome[1]);
    free(frac -> genome[2]);
    free(frac -> genome[3]);
    free(frac -> genome);
    return;
}

void freefrac(struct Fractal *frac){
    /* This function frees the memory of a fractal structure */
    for (int i = 0; i < HEIGHT; i++){
        free(frac -> bm[i]);
    }
    free(frac -> bm);
    freegenome(frac);
    free(frac -> xs);
    free(frac -> ys);
    return;
}

int lenfile(char *filename){
    /* Thise function calculates the length of a file */
    int len = 0;
    char ch;
    FILE *fp = fopen(filename, "r");
    if (fp == NULL){
        fprintf(stderr, "Failed␣to␣open␣file␣(lenfile):␣%s␣(%c)\n", filename, 'r');
        exit(1);
    }
    while (!feof(fp)){
        ch = fgetc(fp);
        if (ch == '\n'){
            len++;
        }
    }
    fclose(fp);
    return len;
}

/*Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: fracfuncs.h
 */
struct Fractal;
struct Fractal * makerandfrac(int numpoints, int numfuncs, int *restrictions, int
    numrestrictions, int disperse, double *window);
void dimension(struct Fractal *frac);
void stddev(struct Fractal *frac);

/* Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: fracfuncs.c
 *
 * This file contains functions that are used to
 * calculate properties from fractals as well as
 * manipulate / mutate them for genetic algorithms
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "Fractals.h"
#include "fracfuncs.h"

struct Fractal * makerandfrac(int numpoints, int numfuncs, int *restrictions, int
    numrestrictions, int disperse, double *window){
    /* This function generates a random fractal. See Fractals.c -> generategenome() for an
     * explanation of the input parameters
     */
    struct Fractal *frac;
    if ((frac = (struct Fractal *)malloc(sizeof(struct Fractal))) == NULL){
                fprintf(stderr, "Malloc failed. (makerandfrac)\n");
                exit(1);
        }
    initializefrac(frac, numfuncs, numpoints);
    //frac -> coloured = 0;
    generategenome(frac, restrictions, numrestrictions, disperse);
    generatefrac(frac);
    generatematrix(frac, window);
    return frac;
}
```

```
void dimension(struct Fractal *frac){
    /* This function calculates an estimate of the
     * fractal dimension for a fractal, based on its
     * corresponding number of pixels, and stores it
     * in the fractal struct.
     */
    frac -> dimension = (log(((double)frac -> numb))/log(((double)WIDTH)));
    return;
}

void stddev(struct Fractal *frac){
    /* This function calculates the standard deviation of the
     * pixels corresponding to a fractal, in both the x and y
     * direction, based on the image of the fractal. It then
     * stores these values in the fractal struct.
     */
    int i,j;
    double stddevx = 0;
    double stddevy = 0;
    for (i = 0; i < HEIGHT; i++){
        for (j = 0; j < WIDTH; j++){
            if ((frac -> bm[i][j]) == 0){
                stddevx += (j - frac -> avgx) * (j - frac -> avgx);
                stddevy += (i - frac -> avgy) * (i - frac -> avgy);
            }
        }
    }
    frac -> stddevx = sqrt(stddevx/((double)(frac -> numb -1)));
    frac -> stddevy = sqrt(stddevy/((double)(frac -> numb -1)));
    return;
}

/*Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: vecio.h
 */
void dstrtovec(char *str, double *vector, int *len);
void istrtovec(char *str, int *vector, int *len);
int * ivecmem(int m);
double ** dmatmem(int m, int n);
void dsortvec(int m, double *vec);
void dsortmatrows(int m, int p, int col, double **mat);

/*Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: vecio.c
 *
 * This file contains some simple functions
 * for vector manipulation
 */
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void dstrtovec(char *str, double *vector, int *len){
    char *ptr;
    *len = 0;
    ptr = strtok(str, ",");
    while (ptr != NULL) {
        vector[*len] = atof(ptr);
        ptr = strtok(NULL, ",");
        *len += 1;
    }
}

void istrtovec(char *str, int *vector, int *len){
    char *ptr;
    *len = 0;
    ptr = strtok(str, ",");
    while (ptr != NULL) {
        vector[*len] = atof(ptr);
        ptr = strtok(NULL, ",");
        *len += 1;
    }
}

int * ivecmem(int m) {
    int *vec;
    vec = (int *)malloc(m * sizeof(int));
    return vec;
}

double ** dmatmem(int m, int n){
    int i;
    double **A;
    A = (double **)malloc(m * sizeof(double *));
    for (i=0;i<m;i++){
            A[i] = (double *)malloc(n * sizeof(double));
    }
    return A;
}

void dsortvec(int m, double *vec){
    /* Sorts a list from lowest to heighest */
    int i, swapped, n, tmp;
    for (i = 0; i < m; i++){
        swapped = 0;
        n = 0;
        while (n < (m - 1)){
            if (vec[n] > vec[n+1]){
                tmp = vec[n];
                vec[n] = vec[n+1];
                vec[n+1] = tmp;
                swapped = 1;
            }
            n += 1;
        }
        if (swapped == 0) break;
    }
    return;
}

/* Sorting function for ordergenome in Fractals.c */
void dsortmatrows(int m, int p, int col, double **mat){
    /* Sorts a matrix by from smallest to largest in column col */
    int i, swapped, n;
    double *tmp;
    for (i = 0; i < m; i++){
        swapped = 0;
        n = 0;
        while (n < (m - 1)){
            if (mat[n][col] > mat[n+1][col]){
                tmp = mat[n];
                mat[n] = mat[n+1];
                mat[n+1] = tmp;
                swapped = 1;
            }
            else if (mat[n][col] == mat[n+1][col]){
                if ((col + 1) <= p) dsortmatrows(m, p, col + 1, mat);
            }
            n += 1;
        }
        if (swapped == 0) break;
    }
    return;
}

/*Created by:  Liam Graham
 * Date:         Oct. 2018
 * Last updated: Jun. 2020
 *
 * FILE NAME: PNGio.h
 */

struct Fractal;
void funcnumtocolours(int colour, int *r, int *g, int *b);
void WritePNG(char *filename, struct Fractal *frac);

/* Created by:  Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME PNGio.c
 *
 * This file contains functions that are used to
 * for fractal image input and output. Specifically,
 * to convert pixel maps of fractals to a png, either
 * coloured or not
 */

#include <stdio.h>
#include <stdlib.h>
#include <png.h>
#include <math.h>
#include "PNGio.h"
#include "Fractals.h"

void funcnumtocolours(int colour, int *r, int *g, int *b){
    /* This function is used to convert a function number
     * to a specific colour so that you can visualize which
     * function made what part of a fractal
     *
     * function   0:     red
     *            1:     orange
     *            2:     yellow
     *            3:     green
     *            4:     blue
     * add more colours if you want to colour an attractor
     * that has more than 4 functions in its IFS
     */

    if (colour == 0){
        *r = 255;
        *g = 0;
        *b = 0;
    }
    else if (colour == 1){
        *r = 255;
        *g = 128;
        *b = 0;
    }
    else if (colour == 2){
        *r = 255;
        *g = 255;
        *b = 0;
    }
    else if (colour == 3){
        *r = 0;
        *g = 255;
        *b = 0;
    }
    else if (colour == 4){
        *r = 0;
        *g = 128;
        *b = 255;
    }
    return;
```

```
}

void WritePNG(char *filename, struct Fractal *frac){
    /* This function converts a pixel map of a fractal,
     * stored in the fractal structure, to a png image
     * stored in the given filename.
     *
     * The fractal structure also contains an integer
     * named coloured, if coloured is 0, the fractal is
     * coloured based on which function output what point
     * according to the colours assigned in funcnumtocolours.
     * If coloured is 1 then the fractal is black.
     */
    FILE *fp = fopen(filename, "wb");
    if (!fp) abort();
    png_structp png = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png) abort();
    png_infop info = png_create_info_struct(png);
    if (!info) abort();
    if (setjmp(png_jmpbuf(png))) abort();
    png_init_io(png, fp);
    png_set_IHDR(
        png,
        info,
        WIDTH,
        HEIGHT,
        8,
        PNG_COLOR_TYPE_RGB,
        PNG_INTERLACE_NONE,
        PNG_COMPRESSION_TYPE_DEFAULT,
        PNG_FILTER_TYPE_DEFAULT
    );
    png_write_info(png, info);
    png_bytep *row_pointers = (png_bytep *)malloc(HEIGHT * sizeof(png_bytep));
    for (int i = 0; i < HEIGHT; i++){
        row_pointers[i] = (png_bytep)malloc(3 * WIDTH * sizeof(unsigned char));
    }
    int r,g,b;
    for (int i = 0; i < HEIGHT; i++){
        for (int j = 0; j < WIDTH; j++){
            if (frac -> bm[i][j] != 255 && frac -> coloured == 0){
                funcnumtocolours(frac -> bm[i][j], &r, &g, &b);
                row_pointers[i][3*j+0] = (unsigned char) r;
                row_pointers[i][3*j+1] = (unsigned char) g;
                row_pointers[i][3*j+2] = (unsigned char) b;
            }
            else if (frac -> bm[i][j] != 255 && frac -> coloured == 1){
                row_pointers[i][3*j+0] = (unsigned char) 0;
                                row_pointers[i][3*j+1] = (unsigned char) 0;
                                row_pointers[i][3*j+2] = (unsigned char) 0;
            }
            else {
                row_pointers[i][3*j+0] = (unsigned char) 255;
                row_pointers[i][3*j+1] = (unsigned char) 255;
                row_pointers[i][3*j+2] = (unsigned char) 255;
            }
        }
    }
    png_write_image(png, row_pointers);
    png_write_end(png, NULL);
    fclose(fp);
    if (png && info) png_destroy_write_struct(&png, &info);
    return;
}

/* FILE NAME: matvec_read.h */
double **matrix_read(char *filename, int *rows, int *cols);

/* This file was created by Allan Willms
 * and is being used with his permission
 *
 * FILE NAME: matvec_read.c
 *
 * The function in this file reads in a matrix
 * stored in the file filename, and calculates
 * the number of rows and columns as it does so
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "matvec_read.h"

double **matrix_read(char *filename, int *rows, int *cols) {
   /* Read a text file that defines a matrix.
    * Each row should have the same number of entries as the first row.
    * Additional entries are ignored; missing entries are treated as zero.
    * Entries in a row are separated by whitespace or commas.
    *
    * Arguments:
    *    input file name,
    *    pointer to int to store number of rows
    *    pointer to int to store number of columns
    *
    * Returns:
    *    NULL pointer on failure
    *    Pointer to pointer to double otherwise.  The matrix is
    *       stored contiguously in memory.
    */

   int m,n,rowlen;
   double **matrix;
   char b;
   char *line,*tok;
   char delimiters[5] = ", \t\n";
```

```
FILE *infile;

if ((infile = fopen(filename,"r")) == NULL) {
   fprintf(stderr,"Failed to open file %s.\n",filename);
   return NULL;
}
/* First read characters until the first newline character is encountered. */
rowlen = 0;
b = fgetc(infile);
while (b != EOF && b != '\n') {
   rowlen++;
   b = fgetc(infile);
}
rowlen += 2;
rewind(infile);
/* Allocate space for, and read the first line into a string. */
line = (char *) malloc(rowlen*sizeof(char));
fgets(line,rowlen,infile);
/* Count the number of entries in this line. */
tok = strtok(line,delimiters);
*cols = 0;
while (tok != NULL) {
   (*cols)++;
   tok = strtok(NULL,delimiters);
}
free(line);
/* Count the remaining lines, keeping track of the longest line. */
*rows = 1;
n = 0;
b = fgetc(infile);
while (b != EOF) {
   if (b == '\n') {
      (*rows)++;
      if (n > rowlen) rowlen = n;
   }
   else
      n++;
   b = fgetc(infile);
}
/* Allocate space for longest line. */
line = (char *) malloc((rowlen+2)*sizeof(char));
/* Allocate space for the row pointers. */
if ((matrix = (double **) malloc(*rows*sizeof(double *))) == NULL) {
   fprintf(stderr,"malloc failed\n");
   fclose(infile);
   return NULL;
}
/* Rewind file and read lines. */
rewind(infile);
/* Allocate space for all rows in contiguous region in memory. */
if ((matrix[0] = (double *) calloc((*rows)*(*cols),sizeof(double))) == NULL) {
   fprintf(stderr,"calloc failed\n");
```

```
      fclose(infile);
      return NULL;
   }
   /* Read each line of file and record values. */
   for (m=0; m<*rows; m++) {
      /* Assign the row pointer to the appropriate location in the allocated space. */
      matrix[m] = matrix[0] + *cols*m;
      /* Get the line from the file. */
      if (fgets(line,rowlen,infile) == NULL) {
         fprintf(stderr,"Error reading input file line %d.\n",m+1);
         fclose(infile);
         return NULL;
      }
      /* Break the line by white space. */
      tok = strtok(line,delimiters);
      for (n=0; n<*cols; n++) {
         if (tok == NULL) break;  /* missing columns treated as zeros */
         if (sscanf(tok,"%lf",matrix[m]+n) != 1) {
            fprintf(stderr,"Error on line %d column %d of input file %s:%s%s.\n",
                  m+1,n+1,filename," Trying to read: ",tok);
            fclose(infile);
            return NULL;
         }
         tok = strtok(NULL,delimiters);
      }
   }
   free(line);
   fclose(infile);
   return matrix;
}

/*Created by:    Liam Graham
 * Last updated: Jun. 2020
 *
 * FILE NAME: generatedata.c
 *
 * This is the main file that, when run,
 * generates databases of fractals
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include "Fractals.h"
#include "vecio.h"
#include "PNGio.h"
#include "fracfuncs.h"

int main(int argc, char *argv[]){
    int i, j, numpoints, numfuncs, numrows, numtogenerate, numrestrictions, disperse, tmpint
        ;
```

```
int *restrictions = ivecmem(20);
int pcomp = 0;
double window[4];
char filename[50], dirname[50], fracname[124],filepath[100],tmp[50];
FILE *fp;
struct Fractal *frac;
srand(time(NULL));

fprintf(stdout, "How␣many␣fractals␣would␣you␣like␣to␣generate:␣");
scanf("%d", &numtogenerate);
fprintf(stdout, "\n");
sprintf(filename, "fracdata.dat");
fprintf(stdout, "What␣is␣the␣directory␣called␣(Note:␣it␣should␣already␣be␣created):␣");
scanf("%s", dirname);
fprintf(stdout, "\nHow␣many␣points␣would␣you␣like␣to␣plot␣for␣each␣fractal:␣");
scanf("%d", &numpoints);
strcpy(filepath, dirname);
fprintf(stdout, "\nHow␣many␣functions␣in␣each␣IFS:␣");
scanf("%d", &numfuncs);
fprintf(stdout, "\nEnter␣a␣vector␣representing␣the␣viewing␣window␣(eg.␣minx,maxx,miny,
    maxy):␣");
scanf("%s", tmp);
dstrtovec(tmp, window, &tmpint);
fprintf(stdout, "\n0␣␣-␣Affine\n");
fprintf(stdout, "1␣␣-␣x␣->␣acos(bx)␣+␣ccos(dy)+e\n");
fprintf(stdout, "2␣␣-␣x␣->␣acos(bx)␣+␣csin(dy)+e\n");
fprintf(stdout, "3␣␣-␣x␣->␣acos(bx)␣+␣ctanh(dy)+e\n");
fprintf(stdout, "4␣␣-␣x␣->␣asin(bx)␣+␣ccos(dy)+e\n");
fprintf(stdout, "5␣␣-␣x␣->␣asin(bx)␣+␣csin(dy)+e\n");
fprintf(stdout, "6␣␣-␣x␣->␣asin(bx)␣+␣ctanh(dy)+e\n");
fprintf(stdout, "7␣␣-␣x␣->␣atanh(bx)␣+␣ccos(dy)+e\n");
fprintf(stdout, "8␣␣-␣x␣->␣atanh(bx)␣+␣csin(dy)+e\n");
fprintf(stdout, "9␣␣-␣x␣->␣atanh(bx)␣+␣ctanh(dy)+e\n");
fprintf(stdout, "10␣-␣Piecewise␣Affine\n");
fprintf(stdout, "\nEnter␣a␣vector␣containing␣the␣maps␣you␣would␣like␣to␣restrict:␣");
scanf("%s", filepath); //filepath is just a temporary placeholder
fprintf(stdout, "\n");
istrtovec(filepath, restrictions, &numrestrictions);
fprintf(stdout, "\n0␣-␣No␣restrictions\n");
fprintf(stdout, "1␣-␣Ensure␣there␣are␣at␣least␣2␣different␣transformation␣types\n");
fprintf(stdout, "2␣-␣Ensure␣there␣is␣at␣least␣1␣of␣each␣transformation␣type\n");
fprintf(stdout, "\nWhat␣dispersion␣of␣transformations␣would␣you␣like:␣");
scanf("%d", &disperse);
fprintf(stdout, "\n");
sprintf(filepath, "%s%s", dirname, filename);
if ((fp = fopen(filepath, "r")) == NULL){
    numrows = 0;
}
else {
    numrows = lenfile(filepath);
}
if ((fp = fopen(filepath, "a")) == NULL){
    fprintf(stderr, "Error, you must create the directory first\n");
    exit(1);
}
fprintf(stdout, "Generating fractals %d to %d\n", numrows, numrows+numtogenerate);
for (i = 0; i < numtogenerate; i++){
    frac = makerandfrac(numpoints, numfuncs, restrictions, numrestrictions, disperse,
        window);
    stddev(frac);
    dimension(frac);
    int params = 0;
    /* fractal number, numfuncs, numpoints, numb, avgx, avgy, stddevx, stddevy,
        dimension, genome */
    fprintf(fp, "%d\t%d\t%d\t%d\t%d\t%d\t%.15lf\t%.15lf\t%.15lf\t",
            numrows+i, frac->numfuncs, frac->numpoints, frac->numb,
            frac->avgx, frac->avgy, frac->stddevx, frac->stddevy, frac -> dimension);
    for (j = 0; j < frac -> numfuncs; j++){
        for (int k = 0; k < multindjump(frac->genome[3][j]); k++){
            fprintf(fp, "%.15lf\t", frac -> genome[0][params + k]);
        }
        params += multindjump(frac->genome[3][j]);
    }
    params = 0;
    for (j = 0; j < frac -> numfuncs; j++){
        for (int k = 0; k < addindjump(frac->genome[3][j]); k++){
            fprintf(fp, "%.15lf\t", frac -> genome[1][params + k]);
        }
        params += addindjump(frac->genome[3][j]);
    }
    for (j = 0; j < frac -> numfuncs; j++){
        fprintf(fp, "%.15lf\t", frac -> genome[2][j]);
    }
    for (j = 0; j < frac -> numfuncs-1; j++){
        fprintf(fp, "%.15lf\t", frac -> genome[3][j]);
    }
    fprintf(fp, "%.15lf\n", frac -> genome[3][frac -> numfuncs -1]);
    sprintf(fracname, "%sfrac%d.png", dirname, numrows+i);
    WritePNG(fracname, frac);
    freefrac(frac);
    if (numtogenerate >= 100 && (i%((int)(numtogenerate/100.)) == 0)){
        pcomp += 1;
        if (pcomp < 10){
            fprintf(stdout, "\rPercent Complete:\t%d%%", pcomp);
        }
        else if (pcomp < 100){
            fprintf(stdout, "\rPercent Complete:\t\b%d%%", pcomp);
        }
        else {
            fprintf(stdout, "\rPercent Complete:\t\b\b%d%%",pcomp);
        }
        fflush(stdout);
    }
}
fprintf(stdout, "\n");
fclose(fp);
exit(0);
}
```

To compile this code, one can run the following in the terminal:

```
$ gcc -Wall -o generatedata generatedata.c Fractals.c fracfuncs.c PNGio.c vecio.c
    matvec_read.c -lm -lpng
```

An example of running the compiled code to create a 100 fractal database of IFSs created with the functions $g_1 = g_3 = tanh(x)$ and $g_2 = g_4 = sin(x)$ is given below.

```
$ mkdir Test
$ ./generatedata
How many fractals would you like to generate: 100

What is the directory called (Note: it should already be created): ./Test/

How many points would you like to plot for each fractal: 1000000

How many functions in each IFS: 4

Enter a vector representing the viewing window (eg. minx,maxx,miny,maxy): -3,3,-3,3

0  - Affine
1  - x -> acos(bx) + ccos(dy)+e
2  - x -> acos(bx) + csin(dy)+e
3  - x -> acos(bx) + ctanh(dy)+e
4  - x -> asin(bx) + ccos(dy)+e
5  - x -> asin(bx) + csin(dy)+e
6  - x -> asin(bx) + ctanh(dy)+e
7  - x -> atanh(bx) + ccos(dy)+e
8  - x -> atanh(bx) + csin(dy)+e
9  - x -> atanh(bx) + ctanh(dy)+e
10 - Piecewise Affine

Enter a vector containing the maps you would like to restrict: 0,1,2,3,4,5,6,7,9,10


0 - No restrictions
1 - Ensure there are at least 2 different transformation types
2 - Ensure there is at least 1 of each transformation type

What dispersion of transformations would you like: 0

Generating fractals 0 to 100
Percent Complete:     100%
$
```

Note that this code will only create data, not overwrite it. In the above example it generated fractals 0 to 100. If we were to run it again with the exact same inputs, it would create fractals 100 to 200. The purpose of this was to make it easier to generate large databases. We recommend not generating more than 10,000 at a given time.

B.2 Neural Network Implementation

Below we provide code from four seperate files that were used to create and train the neural networks on the fractal databases. These programs are written in version 3.7 of the Python programming language, and make use of the matplotlib, Pytorch, and numpy libraries.
In the file FracDataset.py:

```
# This file contains the class definition of the dataset
# Created by:   Liam Graham
# Last Updated: June 2020

import os
import numpy as np
import torch
import torch.utils.data as data_utils
from PIL import Image

class FractalDataset(data_utils.Dataset):
    # The class definition of the dataset - this is not to be
    # confused with the data loader which segments the data into
    # batches
    #
    # NOTE: All images of an N-function IFS were stored in one
    #       directory along with a datafile containing their
    #       properties. Each fractal was stored as frac{}.png
    #       where {} is its creation number and corresponds to
    #       the line number in the data file. In the data file
    #       the data was stored in the order of:
    #       - creation number
    #       - number of functions in the IFS
    #       - number of points
    #       - number of pixels corresponding to the fractal
    #       - x coordinate of centroid
    #       - y coordinate of centroid
    #       - standard deviation in the x direction
    #       - standard deviation in the y direction
    #       - fractal dimension estime
    #       - IFS parameters
    #
    # INITIALIZATIONS:
    #     filename:  the name of the datafile
    #     root_dir:  the directory containing the images and datafile
    #     invert:    if 0, load images normally, else, invert the images
    #     transform: a transformation that can be applied to the data as
    #                it is loaded
    def __init__(self, filename, root_dir, invert = 0, transform=None):
        fracdata = np.loadtxt(filename)
        numfuncs = int((len(fracdata[0,:])-9)/7)
        self.outputs = fracdata[:, 9:9+6*numfuncs]
        self.len = len(self.outputs)
        self.root_dir = root_dir
        self.transform = transform
        self.invert = invert

    # returns the amount of elements in the dataset
    def __len__(self):
        return self.len

    # returns an item linenum of the dataset as a python dictionary
    # containing the image of an attractor and its IFS parameters
    def __getitem__(self, linenum):
        img_name = os.path.join(self.root_dir,
                                "frac{}.png".format(linenum))
        image = (Image.open(img_name)).getdata()

        data = self.outputs[linenum, :]
        sample = {'image': image, 'data': data}

        if self.transform:
            sample = self.transform(sample, self.invert)

        return sample

class ToTensor(object):
    # The class definition of the transform the converts
    # the data to a pytorch tensor object

    def __call__(self, sample, invert):
        data = torch.Tensor(sample['data'])
        image = sample['image']
        image = (torch.Tensor(image)).reshape(-1, 640)
        image.unsqueeze_(0)

        if (invert != 0):
            image = -1*(image - 255)

        return {'data': data,
                'image': image}
```

In the file Networks.py:

```
# This file contains the definition of the network used to predict IFS parameters
# Created by:   Liam Graham
# Last Updated: June 2020

import torch
import torch.nn as nn
import os
import numpy as np

def convbr(in_channels, out_channels, kernel_size = 3, stride = 1, padding = 0):
    # Pre-defined concvolutional layer that incorporates batch normalization and
    # the leaky-ReLU activation function
    #
    # INPUTS:
    #     in_channels:  the number of feature maps of the input
    #     out_channels: the number of output feature maps
    #     kernel_size:  the width or height of the kernel in the convolutional layer
    #     stride:       the stride of the kernel
    #     padding:      the number of layers of zero padding
    #
    # OUTPUTS:
    #     conv:         a convolutional layer object

    conv = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size = kernel_size, stride = stride,
            padding = padding),
        nn.BatchNorm2d(out_channels),
        nn.LeakyReLU())
    return conv

def fcbrd(in_channels, out_channels, dropout = 0):
    # Pre-defined fully connected layer that incorporates batch normalization, the
    # leaky-ReLU activation function, and dropout (though it is not used)
    #
    # INPUTS:
    #     in_channels:  the length of the input vector
    #     out_channels: the length of the output vector
    #     dropout:      the dropout probability
    #
    # OUTPUTS:
    #     fc:           a fully connected layer object
    fc = nn.Sequential(
        nn.Dropout(p = dropout),
        nn.Linear(in_channels, out_channels),
        nn.BatchNorm1d(out_channels),
        nn.LeakyReLU()
    )
    return fc

class block(nn.Module):
    # The block class definition used in the neural network
    #
    # INITIALIZATIONS:
    #     channels: the number of features maps of the input and output layers
    #     reduce:   the number of feature maps of the middle layer
    #     ksize:    the size of the kernel in the middel layer
    def __init__(self, channels, reduce, ksize):
        super(block, self).__init__()
        self.conv1 = convbr(channels, reduce, kernel_size = 1)
        self.conv2 = convbr(reduce, reduce, kernel_size = ksize, padding = int(ksize/2))
        self.conv3 = convbr(reduce, channels, kernel_size = 1)

    #Passing the data forward through the block
    #
    # INPUTS:
    #     x: the output of the previous layer
    #
    # OUTPUTS:
    #     out: the output of the block

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out += x
        return out

class FracNet(nn.Module):
    # The class definition of the neural network used to model IFS parameters
    #
    # INITIALIZATIONS:
    #     num_classes: the number of outputs of the network
    #     dropout:     the dropout probability (this was not used)
    #     chunks:      a list of size 7 representing the chunk sizes
    #     ksize:       a list of size 7 representing the kernel size
    #                  for the layer in each chunk
    #                  note: ksize should only be odd numbers
    #     psize:       a list of size 7 representing the kernel size
    #                  in each pooling layer
    #                  note: psize should only be even numbers

    def __init__(self, num_classes, dropout, chunks, ksize, psize):
        # ATTRIBUTES:
        #     Accuracies:   a numpy array keeping track of the accuracy at
        #                   tolerance levels 0.1, 0.2, 0.3, 0.4, 0.5,
        #                   measured after each epoch
        #     losses:       a numpy array keeping track of the loss
        #                   at each step
        #     total_epochs: the total epochs for which the model has been
        #                   trained
        #     num_maps:     the number of feature maps in a given chunk

        super(FracNet, self).__init__()
        self.Accuracies = [[0],[0],[0],[0],[0]]
        self.losses = np.array([])
        self.total_epochs = 0
        self.num_classes = num_classes
        num_maps = [8,16,32,64,128,256,512]
    #the dimensions after each chunk:
    #640 x 640
    self.pool0 = convbr(1, num_maps[0], kernel_size = psize[0], stride = 2, padding =
        int(psize[0]/2)-1)
    #320 x 320
    self.chunk0 = self.make_chunk(num_maps[0], chunks[0], int(num_maps[0]/4),ksize[0])

    #320 x 320
    self.pool1 = convbr(num_maps[0], num_maps[1], kernel_size = psize[1], stride = 2,
        padding = int(psize[1]/2)-1)
    #160 x 160
    self.chunk1 = self.make_chunk(num_maps[1], chunks[1], int(num_maps[1]/4),ksize[1])

    #160 x 160
    self.pool2 = convbr(num_maps[1], num_maps[2], kernel_size = psize[2], stride = 2,
        padding = int(psize[2]/2)-1)
    #80 x 80
    self.chunk2 = self.make_chunk(num_maps[2], chunks[2],int(num_maps[2]/4),ksize[2])

    #80 x 80
    self.pool3 = convbr(num_maps[2], num_maps[3], kernel_size = psize[3], stride = 2,
        padding = int(psize[3]/2)-1)
    #40 x 40
    self.chunk3 = self.make_chunk(num_maps[3], chunks[3], int(num_maps[3]/4),ksize[3])

    #40 x 40
    self.pool4 = convbr(num_maps[3], num_maps[4], kernel_size = psize[4], stride = 2,
        padding = int(psize[4]/2)-1)
    #20 x 20
    self.chunk4 = self.make_chunk(num_maps[4], chunks[4], int(num_maps[4]/4),ksize[4])

    #20 x 20
    self.pool5 = convbr(num_maps[4], num_maps[5], kernel_size = psize[5], stride = 2,
        padding = int(psize[5]/2)-1)
    #10 x 10
    self.chunk5 = self.make_chunk(num_maps[5], chunks[5], int(num_maps[5]/4),ksize[5])

    #10 x 10
    self.pool6 = convbr(num_maps[5], num_maps[6], kernel_size = psize[6], stride = 2,
        padding = int(psize[6]/2)-1)
    #5 x 5
    self.chunk6 = self.make_chunk(num_maps[6], chunks[6], int(num_maps[6]/4),ksize[6])

    self.fc0 = fcbrd(num_maps[6] * 5 * 5, 1000, dropout = dropout)
    self.fcout = nn.Linear(1000, num_classes)


def make_chunk(self, channels, chunks, reduce, ksize):
    # This class function creates the chunks in the network by
    # using the block class above
    #
```

```
        # INPUTS:
        #     channels: the number of features maps of the input and output layers
        #               of each block in each chunk
        #     chunks:   the size of the chunk to be made
        #     reduce:   the number of feature maps of the middle layer of each block
        #     ksize:    the size of the kernel in the middel layer of each block
        layers = []
        for i in range(chunks):
            layers.append(block(channels, reduce, ksize))
        return nn.Sequential(*layers)

    def forward(self, x):
        #Passing the data forward through the block
        #
        # INPUTS:
        #     x: the output of the previous layer
        #
        # OUTPUTS:
        #     out: the output of the block

        out = self.pool0(x)
        out = self.chunk0(out)

        out = self.pool1(out)
        out = self.chunk1(out)

        out = self.pool2(out)
        out = self.chunk2(out)

        out = self.pool3(out)
        out = self.chunk3(out)

        out = self.pool4(out)
        out = self.chunk4(out)

        out = self.pool5(out)
        out = self.chunk5(out)

        out = self.pool6(out)
        out = self.chunk6(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc0(out)
        out = self.fcout(out)
        return out
```

In the file Networkfuncs.py:

```
# This file contains functions pertaining to the neural network model
# Created by:   Liam Graham
# Last Updated: June 2020
```

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from Networks import * # The file containing the network class definition
import torchvision
from PIL import Image
import os

def train(model, train_loader, device, optimizer, criterion, printpct = 5, graphpct = 10,
    graphheight = 0.3):
    # This function can be used to train a neural network on multiple GPUs
    #
    # INPUTS:
    #     model:         the network object that will be trained
    #     train_loader:  the object that loads data into batches
    #     device:        the device that the model will be trained on
    #                    Note: this function is for multiple GPUs.
    #                          If training on the CPU or a single GPU,
    #                          change all model.module.* instances to
    #                          model.*
    #     optimizer:     the optimizer that is used in conjunction with
    #                    gradient descent to update network parameters
    #     criterion:     the loss function of the network
    #     printpct:      the percentage frequency that a loss update is
    #                    printed. Eg., 5 prints every 5% completion of
    #                    an epoch
    #     graphpct:      the percentage frequency that a loss update is
    #                    graphed. Eg., 10 graphs the loss every 10%
    #                    completion of an epoch
    #     graphheight:   the height of the graph of the loss printed
    #                    whenever graphpct of an epoch is complete

    model.train()
    total_step = len(train_loader)
    print ("Beginning␣Training")
    model.module.total_epochs += 1
    for i, sample in enumerate(train_loader):
        # Move batch to the device
        images = sample['image'].to(device)
        labels = sample['data'].to(device)

        # Feeding the input forward
        outputs = model(images)
        loss = criterion(outputs, labels)
        model.module.losses = np.append(model.module.losses, loss.item())

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # print an update every printpct percentage completion of an epoch
        if (i+1) % int(printpct*total_step/100) == 0:
            print ('Step_[{}/{}],_Loss:_{:.4f}'
                   .format(i+1, total_step, loss.item()))

        # graph the loss every graphpct percentage completion of an epoch
        if (i+1) % int(graphpct*total_step/100) == 0:
            # Note: the division by 2500 is due to a dataset of size 250,000
            #       with a step size of 100
            xs = np.linspace(0, len(model.module.losses)/2500, len(model.module.losses))
            plt.plot(xs, model.module.losses)
            plt.ylim(0,graphheight)
            plt.grid(True)
            plt.show()
    print ("Epoch_Complete!")

def test(model, test_loader, device, batch_size, test_items = 10000):
    #This function can be used to test a neural network on a dataset
    #
    # INPUTS:
    #     model:        the network object that will be tested
    #     test_loader: the object that loads into batches
    #     device:       the device that the model will be tested on
    #                   Note: this function is for multiple GPUs.
    #                         If training on the CPU or a single GPU,
    #                         change all model.module.* instances to
    #                         model.*
    #     batch_size:  the batch size of the data loader
    #     test_items:  the number of elements in the test set

    model.eval() # this is to switch batch normalization from moving
                 # average to total average
    test_total = test_items * model.module.num_classes
    total_test_step = len(test_loader)
    with torch.no_grad():
        acc1, acc2, acc3, acc4, acc5 = 0, 0, 0, 0, 0
        tol1, tol2, tol3, tol4, tol5 = 0.1, 0.2, 0.3, 0.4, 0.5
        for i, sample in enumerate(test_loader):
            # move batch to device
            image = sample['image'].to(device)
            data = sample['data'].to(device)

            # feed the data forward
            outputs = model(image)

            # calculate the difference between the true values and outputs
            difs = (abs(outputs[:,:] - data[:,:]))
            acc1 += (difs < tol1).sum().item()
            acc2 += (difs < tol2).sum().item()
            acc3 += (difs < tol3).sum().item()
            acc4 += (difs < tol4).sum().item()
            acc5 += (difs < tol5).sum().item()
            #print percentage completion
            if (i+1) % 1 == 0:
                print ('Step␣[{}/{}]'
                       .format(i+1, total_test_step), end = "\r")

    #append to model accuracy attribute
    model.module.Accuracies[0].append(100*acc1/test_total)
    model.module.Accuracies[1].append(100*acc2/test_total)
    model.module.Accuracies[2].append(100*acc3/test_total)
    model.module.Accuracies[3].append(100*acc4/test_total)
    model.module.Accuracies[4].append(100*acc5/test_total)
    print ("Within␣{}:\t{:.2f}%␣␣[{}/{}]".format(tol1, 100*acc1/test_total, acc1, test_total
        ))
    print ("Within␣{}:\t{:.2f}%␣␣[{}/{}]".format(tol2, 100*acc2/test_total, acc2, test_total
        ))
    print ("Within␣{}:\t{:.2f}%␣␣[{}/{}]".format(tol3, 100*acc3/test_total, acc3, test_total
        ))
    print ("Within␣{}:\t{:.2f}%␣␣[{}/{}]".format(tol4, 100*acc4/test_total, acc4, test_total
        ))
    print ("Within␣{}:\t{:.2f}%␣␣[{}/{}]".format(tol5, 100*acc5/test_total, acc5, test_total
        ))

def save(model, learning_rate, optimizer, device, filename):
    #This function can be used to save a neural network
    #
    # INPUTS:
    #     model:          the network object that will be tested
    #     learning_rate: the current learning rate used
    #     optimizer:      the optimizer that is used in conjunction with
    #                     gradient descent to update network parameters
    #     device:         the device the model is currently on
    #     filename:       the name of the file the model will be saved as
    #                     Note: a common file extension type is .tar

    cpudevice = torch.device('cpu')
    model.to(cpudevice)
    state = {
        'losses': model.losses,
        'num_classes': model.num_classes,
        'Accuracies': model.Accuracies,
        'learning_rate': learning_rate,
        'total_epochs': model.total_epochs,
        'dropout': model.dropout,
        'state_dict': model.state_dict(),
        'optimizer': optimizer.state_dict(),
    }
    torch.save(state, filename)
    model.to(device)

def load(filename device, chunks, ksizes, psizes, scale_lr = 1):
    # This function can be used to load a neural network
    #
    # INPUTS:
    #     filename: the name of the file the model will be saved as
    #     device:   the device the model will be loaded to
    #               Note: this function is for multiple GPUs.
    #                     If training on the CPU or a single GPU,
    #                     change all model.module.* instances to
    #                     model.*
    #     The following must match that of the saved model:
    #     chunks:   a list of size 7 representing the chunk sizes
    #     ksize:    a list of size 7 representing the kernel size
    #               for the layer in each chunk
    #               note: ksize should only be odd numbers
    #     psize:    a list of size 7 representing the kernel size
    #               in each pooling layer
    #               note: psize should only be even numbers
    #
    #     scale_lr: the factor by which the learning rate will be
    #               scaled

    checkpoint = torch.load(filename, map_location = 'cpu')
    num_classes = checkpoint['num_classes']
    dropout = checkpoint['dropout']
    model = FracNet(num_classes, dropout, chunks, ksizes, psizes)
    if torch.cuda.device_count() > 1:
        model = nn.DataParallel(model)
    model.to(device)
    learning_rate = checkpoint['learning_rate']*scale_lr
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    model.module.losses = checkpoint['losses']
    model.module.Accuracies = checkpoint['Accuracies']
    model.module.total_epochs = checkpoint['total_epochs']
    model.module.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    optimizer.param_groups[0]['lr']*= scale_lr
    print ("Model_loaded_succesfully:\tEpoch:",model.module.total_epochs,"loaded_onto",
        device)
    return model, optimizer, learning_rate
```

To create a model and train it we can run the following Python script:

```
# This file contains can be used to load data, create a network, train the model, and save
    it
# Created by:   Liam Graham
# Last Updated: June 2020

import os
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from PIL import Image
```

```
import torch.utils.data as data_utils
import matplotlib.pyplot as plt
import numpy as np
from FracDataset import FractalDataset, ToTensor # The file containing the Fractal dataset
    class definition
from Networks import *       # The file containing the network class definition
from Networkfuncs import *   # The file containing the network function such as train, test,
     etc.

if __name__ == "__main__":
    batch_size  = 100
    num_workers = 6    # The number of cpus used to load data

    # Loading the Fractal dataset
    # This file will load the training set from the directory
    # /home/lgraha07/scratch/Fraclibnf2/
    #
    # The validation set from
    # /home/lgraha07/scratch/ValidFraclibnf2/
    #
    # and the test set from
    # /home/lgraha07/scratch/TestFraclibnf2/

    dirname = "Fraclibnf2"
    testdirname = "TestFraclibnf2"
    validdirname = "ValidFraclibnf2"

    path = "/home/lgraha07/scratch/"

    dirpath = path+dirname
    testdirpath = path+testdirname
    validdirpath = path+validdirname

    filename = "/fracdata.dat"
    filepath = dirpath+filename
    testfilepath = testdirpath+filename
    validfilepath = validdirpath+filename

    train_set = FractalDataset(filename = filepath,
                               root_dir = dirpath,
                               transform = ToTensor())
    test_set = FractalDataset(filename = testfilepath,
                              root_dir = testdirpath,
                              transform = ToTensor())
    valid_set = FractalDataset(filename = validfilepath,
                                root_dir = validdirpath,
                                transform = ToTensor())

    # Put each dataset into a pytorch data loader which divides it into batches
    frac_train_loader = data_utils.DataLoader(train_set,
                                              batch_size = batch_size,
                                              shuffle = True,
                                              num_workers = num_workers)
    frac_test_loader = data_utils.DataLoader(test_set,
                                              batch_size = batch_size,
                                              shuffle = False,
                                              num_workers = num_workers)
    frac_valid_loader = data_utils.DataLoader(valid_set,
                                              batch_size = batch_size,
                                              shuffle = False,
                                              num_workers = num_workers)

    # Initializations and Hyperparameters
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    num_classes = train_set[0]['data'].size(0)
    dropout = 0
    learning_rate = 0.0000075
    chunks = [3,5,7,11,29,23,17]
    ksizes = [3,3,3,3,3,3,3]
    psizes = [4,4,4,4,4,4,4]
    betas = (0.9,0.999)
    num_epochs = 10

    # Create the model
    model = FracNet(num_classes, dropout, chunks, ksizes, psizes)
    if torch.cuda.device_count() > 1:
        model = nn.DataParallel(model)
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate, betas = betas,
        weight_decay = 0.01)

    # Train, test and save the model. Note: during training it is tested on the validation
        set
    # Once all training is complete is when it is tested on the test set
    for i in range(num_epochs):
        train(model, frac_train_loader, device, optimizer, criterion = nn.MSELoss())
        test(model, frac_val_loader, device, batch_size)
        modelname = "modelnf2_"+str(i)+".tar"
        save(model, learning_rate, optimizer, device, modelname)

    # To go back and load epoch 8 and scale the learning rate by 1/10, we could then run
    model, optimizer, learning_rate = load("modelnf2_8.tar", device, chunks, ksizes, psizes,
         scale_lr = 1/10.)
```