# Polymorphisms and Neural Networks for Image Classification

Rachel Dennis
Advisors: Alex Iosevich and Charlotte Aten

May 2, 2023

## 1 Abstract

Neural networks are a key tool used in machine learning. While neural networks conventionally use continuous activation functions, in this paper we explore the use of discrete activation functions, namely polymorphisms. We define a neighbor function that we use to update the activation functions while training. We also define dominion polymorphisms, which come from a combinatorial object analogous to a graph coloring, and create a dominion activation function which makes the neural network more powerful. We implement and test a neural network with these features and discuss the results.

## 2 Table of Contents

## Contents

# 3 Background

## 3.1 Neural Networks

Neural networks are an essential tool in machine learning and artificial intelligence, with a wide variety of applications. Neural networks consist of layers of nodes through which a set of input data is passed. Each node has an activation function, and the network trains by running a set of input data, comparing it to the expected output, and then making small adjustments to the activation functions. We give a more formal definition as follows:

**Definition 1.** A neural network on a set $A$ consists of a layered graph with vertices $V$ and edges $E$. We label the layers of the graph $V_0, V_1, ..., V_n$. Each vertex $v$ (also called a node) in row $V_i$ has an associated activation function $\Phi : A^k \to A$, where $k$ is the number of nodes in $V_{n-1}$ which are connected to $v$ by an edge.

The first layer of the neural network, $V_0$, takes the input to the network and passes that data to the next layer without passing it through an activation function. Each subsequent layer takes the outputs of previous layer, as well as an extra node, as its inputs. The outputs of the last layer are the outputs of the network. For a network performing classification, the network gives the class of the input data as its only output.

To train a neural network to classify images, we use a set of training data where each data point is an image and its class. For this paper we used images of handwritten digits and the class is the digit which is displayed. During the training process the network attempts to classify the image, then compares the class it assigned to the correct class. If the classes don't match, we use a process called backpropagation to work backwards through the network and make small changes to the activation functions.

If the activation functions are continuous, there is a relatively intuitive understanding of what it means for two functions to be similar. We can make small changes to the constants in the equation to get a function that is "close" to the original function. This works well for the majority of neural networks, which use continuous activation functions such as the the sigmoid function $\frac{1}{1+e^{-x}}$. Another common construction is for each node to have a weight which is multiplied by the sum of the inputs. In this case we make the weight slightly larger or smaller

depending on whether we want that node to have more or less impact on the output of the network.

When using discrete activation functions, the concept of a small change to the function is much less intuitive. In the next section we examine the use of polymorphisms as activation functions and how we can conceptualize a "small" change to a polymorphism.

## 3.2 Polymorphisms and Neighbor Functions

**Definition 2.** A graph homomorphism from $G$ to $H$ is an operation that maps adjacent vertices in $G$ to adjacent vertices in $H$.

**Definition 3.** An n-ary graph polymorphism is a graph homormorphism $h : G^n \to G$.

We use polymorphisms as our activation functions in this project, so we need a way to make small changes to a polymorphism. In order to do this we use what we call neighbor functions that make small changes to a polymorphism. There is no universal definition of what constitutes a small change to a polymorphism, so we have chosen a few changes that can be made to a polymorphism while still generating what we consider similar images.

**Definition 4.** An endomorphism on a graph $G$ is a homomorphism $h : G \to G$.

We make changes to a polymorphism activation function by composing it with an endomorphism. The endomorphisms we use are:

- a rotation by a factor of $\frac{\pi}{2}$
- a reflection about the y axis
- a componentwise sum of the input image with another binary image
- a Hadamard (componentwise) product of the input image with another binary image.

When adjusting the activation functions during training, our program calls on the neighbor function, which generates a set of similar polymorphisms by composing the original polymorphism with the endomorphisms listed above, then randomly picks one of the newly generated polymorphisms to replace the original.

We've discussed how to make changes to a polymorphism, but we have yet to discuss what polymorphisms we'll be using. To do this we need to define Hamming graphs.

## 3.3 Hamming Graphs

A Hamming graph stores tuples of binary images as its vertices.

Let $n \in \mathbb{N}$ and let $A_n = Mat_n(\mathbb{F}_2)$.

**Definition 5.** We call $a \in A_n$ a binary image and say that it has size $n$. We call an entry $a_{ij}$ in the matrix $a$ a pixel.

An $n$-dimensional Hamming graph stores all the binary images of size $n$. To explain the concept of adjacent images in the graph, we define the Hamming distance.

**Definition 6.** The Hamming distance between two binary images $a_1, a_2 \in A_n$ is given by

$$d(a_1, a_2) \coloneqq |\{(i,j) \in [n]^2 | (a_i)_{ij} \neq (a_2)_{ij}\}|$$

. In other words, the Hamming distance is the number of pixels whose values are different in $a_1$ and $a_2$.

Now we can formally define the Hamming graph.

**Definition 7.** The n-dimensional Hamming graph is

$$\mathbf{Ham}_n \coloneqq (A_n, \{(a_1, a_2) \in A_n^2 | d(a_1, a_2) \leq 1\}).$$

The k-ary n-dimensional Hamming graph is

$$\mathbf{Ham}_n^k \coloneqq (A_n^k, \{(a_1, a_2) | a_1, a_2 \in A_n^k \text{ and } d(a_{1_i}, a_{2_i}) = 1 \text{ for at most one } i \in \{1, ..., k\}\}).$$

When constructing the polymorphisms it is useful to simplify the data stored at the vertices. To do this we introduce the concept of a Hamming weight.

**Definition 8.** A binary image $a \in A_n$ has Hamming weight

$$||a|| \coloneqq d(a, 0)$$

where 0 is the binary image where every pixel takes value 0.

**Definition 9.** A Hamming weight map $\psi_k : A_n^k \to [n^2 + 1]^k$ is defined by

$$\psi_k(a_1, ..., a_k) \coloneqq (||a_1||, ..., ||a_k||)$$

.

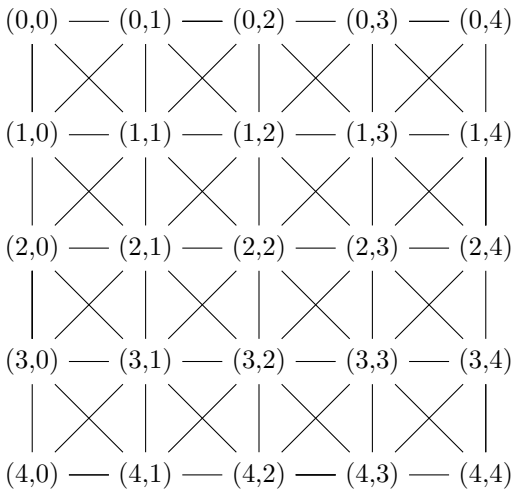
**Definition 10.** The k-ary Hamming weight graph of size n is given by $\mathbf{Ham}_n^k / ker(\psi_k)$.

**Example 1.** Figure 1 shows the binary Hamming weight graph of size 2.

## 3.4 Dominions

We want our activation function to map a tuple of binary images to a single binary image. For the purposes of this paper, we focus on mapping pairs of binary images in order to keep our tests simple. To ensure that our homomorphism maps $\mathbf{Ham}_n^2 / ker(\Psi_n)$ to $\mathbf{Ham}_n$, we use a coloring of the Hamming weight graph to facilitate the process.

Figure 1: The binary Hamming weight graph of size 2




**Definition 11.** We refer to a coloring of of a Hamming weight graph as a dominion.

To construct a homomorphism from $\mathbf{Ham}_n^2/ker(\Psi_n)$ to $\mathbf{Ham}_n$, we need to impose some restrictions on our coloring. A Hamming graph cannot contain any 3-cycles due to its construction. For example, if we try to create a 3-cycle starting with an image where the value of each pixel is 0, we get the following adjacent images:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$
$$\vert$$
$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$
$$\vert$$
$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

For a 3-cycle to exist, there would have to be an edge in the graph between two of the images adjacent to the 0 image. However, each pair of these matrices has different values at two pixels. Thus their Hamming distance is 2 and so there is no edge between them in the Hamming graph.

To ensure that the dominion preserves the structure of the Hamming graph,

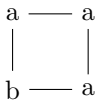
we need to make sure there are no 3-cycles in the dominion. To formalize this constraint we define a basic cube.

**Definition 12.** Given a vertex $v' = (v'_1, v'_2, ..., v'_k)$ of the k-ary Hamming weight graph of size n, the basic cube with $v'$ as its top corner is

$$\{v = (v_0, v_1, ..., v_k) | \forall i, (v_i - v'_i) \in \{0, 1\}\}$$

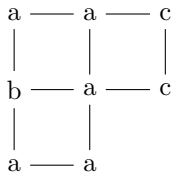
We then restrict the coloring by saying that a dominion can assign at most 2 colors to the vertices of a basic cube.

**Example 2.** We want to define a dominion for the Hamming weight graph shown in Figure 1. Suppose we are using the three colors $\{a, b, c\}$. There are no restrictions on the color of $(0, 0)$, so we assign it $a$. Again, there are no restrictions on the color of $(1, 0)$, so we assign it $b$. Now $(0, 1)$ and $(1, 1)$ are in a basic cube with $(0, 0)$ and $(1, 0)$, so we can only color them $a$ or $b$. We assign them both $a$. Thus we have assigned colors to our first basic cube:




Now we consider the basic cube consisting of $(0, 1)$, $(1, 1)$, $(0, 2)$, and $(1, 2)$. We have already assigned two colors to this cube, so we can assign the remaining vertices only $a$ or $b$. We color $(0, 2)$ $b$ and $(1, 2)$ $a$.

The basic cube consisting of $(1, 0$, $(1, 1)$, $(2, 0)$, and $(2, 1)$ has only been assigned one color, so we can pick any of the three colors for $(2, 1)$. We assign it $c$ and do the same to $(2, 1)$.



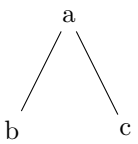

We see that the basic cube consisting of $(1, 1)$, $(1, 2)$, $(2, 1)$, and $(2, 2)$ has only been assigned two colors, so our choices of color work. We continue in this manner to assign colors to all of the vertices in the Hamming weight graph.

To help the program create a dominion, we use a constraint graph to keep track of which colors are adjacent to each other.

**Definition 13.** A constraint graph consists of a set of vertices, which are the colors that we can choose from when creating the dominion, and a set of edges that connect any two colors than can be adjacent to each other in the dominion.

Since there can be at most two colors in any basic cube, there can be no three-cycles in the constraint graph. Thus we can store a constraint graph as a tree.

**Example 3.** The constraint graph for the dominion in Example 2 is




Given a tree and a constraint graph, we can generate a variety of dominions. We define a minimum constraint graph that is specific to each dominion.

**Definition 14.** The minimum constraint graph (MCG) for a dominion includes an edge between two labels iff those two labels are adjacent in the dominion.

**Example 4.** The constraint graph in Example 3 is also a minimum constraint graph because $a, b,$ and $c$ are all used in the dominion in Example 2.

# 4 Implementation

## 4.1 Data Set

We tested our neural networks on the MNIST dataset, a set of images of handwritten numbers between 0 and 9. Each image was 28 pixels by 28 pixels and each pixel had a grayscale value. We converted the images to black-and-white images to simplify our initial tests. Each pixel in the grayscale image had a value between 0 (a black pixel) and 255 (a white pixel). We modified the code from https://www.kaggle.com/code/hojjatk/read-mnist-dataset/notebook to read the MNIST dataset and convert the stored data into images stored as lists of lists. If the original pixel had a value greater than 127 we changed it to a white pixel; otherwise we changed it to a black pixel.

## 4.2 Constructing the Dominion Polymorphism

We want to construct a polymorphism to use as an activation function that uses dominions to map $\mathbf{Ham}_2^2$ to $\mathbf{Ham}_2$. We refer to this function as $g_\alpha$. Given a dominion $D$ and a homomorphism $\alpha$ that maps a label in the dominion to a binary image, we define

$$g_\alpha(a_1, a_2) = \alpha(D(\Psi_2(a_1, a_2))).$$

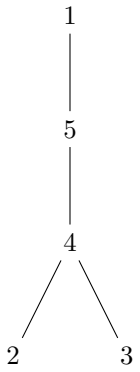
Recall that $\Psi_2$ returns the Hamming weights of $a_1$ and $a_2$.

We store a dominion in python as an array. $D(i, j)$ returns the the label in the i-th row and j-th column of the array. Note that for a binary image of size n, the dimensions of the dominion will be $(n^2 + 1)$x$(n^2 + 1)$.

## 4.3 Constructing Dominions

We want to be able to construct a variety of dominion polymorphisms for any given tree. However, generating a new dominion and homomorphism each time we need a new dominion polymorphism is time-consuming, so we create them in advance and save them. We begin with a set of labels for the coloring and construct a random tree which is our constraint graph.

**Example 5.** To illustrate the algorithms for creating dominions and homomorphisms we'll use a scaled-down version of the problem for 2 by 2 binary images. We start with a set of five colors with labels $\{1, 2, 3, 4, 5\}$ and randomly construct a tree, which will be our constraint graph. For this example we get the following tree:




Once we have set of labels and a constraint graph, we use the following algorithm to generate random dominions. Note that the algorithms are stored as lists of lists. We then save these lists as text files so that we can reconstruct the dominion from the file when creating a dominion polymorphism.

The algorithm to generate a new row is shown below. It works by creating a set of possible labels for the next element in the dominion and a set of labels that cannot be assigned to the next element. If there are two possible labels for the element, one of those two elements must be chosen so that there are no more than two colors in any basic cube. If there is one possible label and one or more rejected labels, we assign the element the one possible label. If there is one possible label and no rejected labels, we can choose any label that is adjacent to the possible label in the constraint graph.

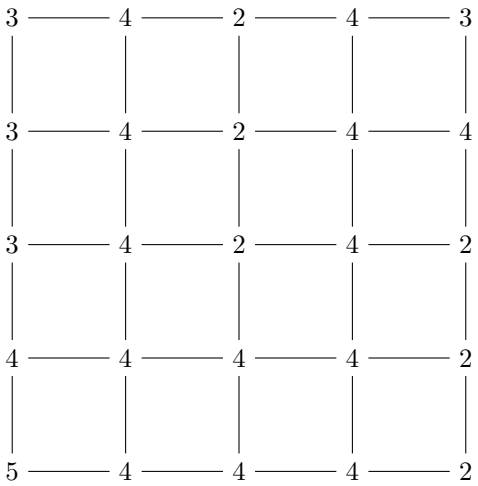
**Example 6.** Since we're using a 2x2 image, we generate a dominion of size 5.

**Algorithm 1** Generate a Random Dominion

**Input:** the size $s$ of the dominion being generated
**Input:** a list of labels $L$
**Input:** a constraint graph, represented as a tree $t$

1: Randomly choose a label from $L$ and make it the first element in a list of lists which we'll call partialDominion.
2: **for** $i = 1$ to $s$ **do**
3: Randomly choose a label that's adjacent to partialDominion[0][i-1] in the constraint graph. (We consider a label to be adjacent to itself.)
4: Assign this label at partialDominion[0][i]
5: **end for**
6: **for** $i = 1$ to $s$ **do**
7: Generate the next line of the graph.
8: **end for**

**Output:** The dominion partialDominion



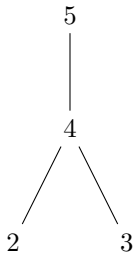

This is a case where the minimum constraint graph is a sugbraph of the constraint graph, since this dominion doesn't include the label 1. The minimum constraint graph is pictured below.

**Algorithm 2** Generate the i-th Row of a Dominion

**Input:** the $(i-1)$-th row $R$ of the dominion
**Input:** a list of labels $L$
**Input:** a constraint graph, represented as a tree $t$

$n =$ the length of $R$
**for** $j = 0$ to $n$ **do**
  Create new sets of nodes, one called candidates and one called rejects. We eventually want candidates to contain all possible labels for the $j$-th element of the new row.
  **if** $j = 0$ **then**
    Add elements 0 and 1 of $R$ to candidates.
  **else if** $j = n - 1$ is the last node in the row **then**
    Add the $(n-1)$-th and $(n-2)$-th elements of $R$ and the the $(n-2)$-th element of the new row to candidates.
  **else**
    Add the $(j)$-th and $(j-1)$-th elements of $R$ to candidates. If the $(j-1)$-th element of the new row matches the $(j)$-th or $(j+1)$-th element of $R$, add it as well.
    Add any of the following elements that aren't already in candidates to rejects: the $(j-1)$-th, $j$-th, and $(j+1)$-th elements of $R$ and the $j-1$-th element of the new row.
  **end if**
  **if** —candidates—=1 and —rejects—=0 **then**
    Let $l$ be the label in candidates. Add any label that's adjacent to $l$ in $t$ to candidates.
  **end if**
  Randomly choose one element of candidates.
  Add the chosen node to the end of the i-th row
**end for**

**Output:** The $i$-th row of the dominion




## 4.4 Constructing Homomorphisms

To construct a homomorphism from a constraint graph $L$ to $\mathbf{Ham}_n$, we map each node in the constraint graph to a binary image. Given an image $a_0$, we call another image $a_1$ adjacent to $a_0$ if $a_0$ and $a_1$ have differing values at at most one pixel.

Starting with a contraint graph represented as a tree, we construct a homomorphism as follows. First assign a random binary image to the root of the tree. Then run the recursion step in Algorithm 3 until every node in the tree has been assigned a binary image.

**Algorithm 3** homomorphismRecursiveStep()

**Input:** a constraint graph $L$
**Input:** $r$, the parent of whichever node is about to be assigned a value
**Input:** the set $N$ of the nodes which are adjacent to $r$ in the tree but have not yet been assigned a binary image
**Input:** $\alpha$, a dictionary containing the values of the homomorphism that have already been assigned

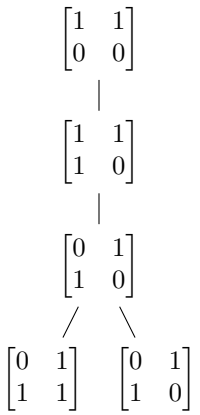
```
1: if N = ∅ then
2:    Return α
3: else
4:    for n ∈ N do
5:       Choose a random binary image that is adjacent to the image assigned to r.
6:       Assign this image as the value n and add the pair as an entry in α.
7:       Let N′ be the set of neighbors of n excluding r.
8:       α = homomorphismRecursionStep(L, n, N′, α)
9:    end for
10: end if
```

**Output:** $\alpha$

**Example 7.** Using the same constraint graph from the previous examples in this section, we want to construct a homomorphism. We begin by assigning the randomly generated binary image $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ at the root of the tree. The only

neighbor of the root is 5. We change the value of the pixel in the second row and first column and assign the new image, $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$, to 5. 4 is the only neighbor of 5 that hasn't been explored. We change the value of the pixel in the first row and first column and assign the new image to 4. Now 4 has two unexplored neighbors, 2 and 3. To get an image for 2 we change the value of the pixel in the second row and second column. Since every binary image is adjacent to itself, we assign the same image to 3 as we did to 4. The complete homomorphism is pictured in Figure 7.

Figure 2: A homomorphism from $L$ to $\mathbf{Ham}_2$

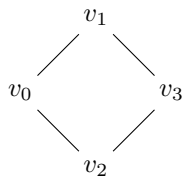


# 5 Testing

## 5.1 Neural Network Construction

To test our neighbor function and $g_\alpha$ we construct a small neural network with one hidden layer. The objective of the network is to classify a binary image by whether the handwritten digit in the image was a 0 or not. Before we began testing we generated a constraint graph with five labels and used that constraint graph to generate twenty dominions and twenty homomorphisms for binary images of size 28. $V_0$, the first layer of the network, contains just one node which takes a binary image as its input. The hidden layer, $V_1$, contains two nodes with unary activation functions, which are initialized as a rotation by $\frac{\pi}{4}$ and a rotation by $\frac{\pi}{2}$. The last layer, $V_2$, contains one node with a binary activation function. We generate a dominion polymorphism using a randomly selected dominion and homomorphism and use it as the initial activation function for this node. The structure of the neural network is pictured in Figure 5.1.

Figure 3: The neural net used to test $g_\alpha$. It consists of three layers: $V_0 = \{v_0\}, V_1 = \{v_1, v_2\},$ and $V_2 = \{v_3\}$.




## 5.2 Training Process

We run the neural network on sets of training MNIST data. Each data point consists of a binary image and a label telling us what handwritten digit the limit depicts. We test the network with a set of 100 training data points and a set of 200 training data points.

The training function takes as an input the number of iterations to run. It will then run the training step that many times. During the training step, the program randomly selects one node from the network and changes its activation function.

We define a loss function which is called during the training step to evaluate our changes to the activation functions. For each element of the training data set, the loss function compares the binary image it expects to see based on the label in the data set to the binary image that it expects to see based on the label it assigns the image after feeding it through the network. The program then returns the number of pixels at which the two images have different values. The loss function returns the average number of pixels where the two images don't have the same value.

Each iteration of the training step calls on the neighbor function described previously to change the activation function. The neighbor function randomly generates a certain number of functions which could be used to replace the activation function. When generating each function, it chooses to either alter the original activation function or replace it entirely. If it alters the activation function, it randomly selects an endomorphism from its list and applies the endomorphism to each input image as well as to the output image.

If it replaces the activation function, it randomly chooses a new activation function based on the arity of the old activation function, or the number of inputs it took. If the arity was one, the program randomly selects a new endomorphism. If the arity was two, the program generates a new dominion polymorphism using a randomly selected dominion and homomorphism. If the arity was any number other than one or two, the program selects an indicator polymorphism.

An indicator polymorphism takes a list of fixed binary images as one of its inputs. It then takes the dot product mod 2 of the list of input images and the list of fixed images. If the dot product is one, the polymorphism returns a white

image with a single black pixel. Otherwise it returns an entirely white image.

The neighbor function returns a list of all the possible replacement functions for the current activation function. It then selects the function with the lowest loss value and makes that the new activation function.

# 6 Results

## 6.1 Improvement in Loss Value

We see that the neural network training correctly due to improvement in the loss function. For both of our training data sets, the loss function decreases until reaching a certain value and then stops decreasing even when we run more iterations of the training step. The training data set with 100 elements converges to a loss of 101.92. The training data set with 200 elements converges to a loss of 82.32.

The fact that the loss function stops decreasing after a certain number of iterations suggests that the algorithm has found the most efficient set of polymorphisms for categorizing the training data. The loss is nonzero, which suggests that the neural network isn't overfitting because it isn't categorizing every function correctly. We also see that the loss decreases when we train with more data points, suggesting that the neural network isn't just predicting the training data but is learning and that it learns better when given more information about the dataset.

The neural network reaches its best loss value quite quickly. For both data sets it consistently achieved the best loss value within 10 trials. In some cases it achieved the best loss value in as little as four iterations for the training set with 200 data points and six iterations for the training set with 100 data points. This is a promising sign that a more complex neural network could be constructed and would still converge within a reasonable time frame.

## 6.2 Next Steps

There is still quite a bit to explore in terms of discrete neural networks. The network that we tested in this paper was still quite simple. We could test more complex neural networks with more hidden layers to see if we get a lower loss value. We could also have the neural network attempt to assign more specific labels to images. Instead of telling us whether or not an image depicts a 0, we would like the network to be able to tell us which of the digits $\{0, 1, 2, ..., 9\}$ is depicted.

We would also like to generalize our dominion polymorphisms to map $\mathbf{Ham}_n^k$ to $\mathbf{Ham}_n$ for cases where $k > 2$.

# 7 Acknowledgements

I would like to thank my advisors, Alex Iosevich and Charlotte Aten, for letting me build on their research on discrete neural networks and for their support and guidance throughout the process. I would also like to thank Juan Rivera-Letelier and Stephen Kleene for being on my thesis committee.

# References

[1] Clifford Bergman. *Universal algebra*. Vol. 301. Pure and Applied Mathematics (Boca Raton). Fundamentals and selected topics. CRC Press, Boca Raton, FL, 2012, pp. xii+308. ISBN: 978-1-4398-5129-6.

[2] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. 32 Avenue of the Americas, New York, NY 10013-2473, USA: Cambridge University Press, 2014. ISBN: 978-1-107-05713-5.