School of Business and Economics (SBE)

BSc. Econometrics & Operations Research

Bachelor Thesis:

Constructions in Combinatorics via Neural Networks

Written by Colin Doumont (`i6207491`)

Supervised by Lars Rohwedder

July 1, 2022

1 Introduction

Within combinatorics, discrepancy theory describes the deviation of a situation from the state one would like it to be in. The discrepancy of a set system in its classical form has been shown to be bound in particular settings. Although these bounds may not hold for variants of discrepancy, no mathematician has yet been able to find a counterexample. The large number of set systems (k^{mn} for m sets and n elements, each taking one of k possible values) makes it impractical to use traditional methods such as human reasoning or computer bruteforcing to identify the few set systems with a high discrepancy.

A radically different approach worth exploring is reinforcement learning, which has seen a boom of success in recent years, with applications ranging from self-driving cars (Pan, You, Wang, & Lu, 2017) to finance (Zhang, Zohren, & Stephen, 2020) and even to notoriously difficult board games, such as Go (Silver et al., 2017) and chess (Mnih et al., 2022). Such games are not very far from combinatorics, in the sense that certain strategic combinations lead to the desired results. In a 2021 preprint, Adam Zsolt Wagner shows that the deep cross-entropy method, a well-known reinforcement-learning technique based on neural networks, could effectively find counterexamples to open conjectures in graph theory and pattern avoidance, two other subfields of combinatorics. Given these encouraging results, we wondered whether Wagner's method can help us find counterexamples to the upper bound of discrepancy variants. As a first step towards answering this question, we investigated whether Wagner's technique can be adapted to find set systems of high discrepancy, whether classical or variant (prefix and fractional in our case). Besides discrepancy theory itself, our research could also advance other fields, such as approximation algorithms, which rely on discrepancy theory for solving certain problems.

The rest of this paper is organized as follows. Section 2 provides the reader with the required background information: an introduction to discrepancy theory, a short description of neural networks, and a high-level summary of Wagner's 2021 paper. Section 3 lays down the details of our methodology, specifically our basic implementation setup and how we improved it to address issues of signal and speed. Section 4 presents our findings for classical, prefix, and fractional discrepancies. Section 5 concludes with the implications of our findings and suggests further research paths.

2 Background

Essentially, we attempted to find set systems of high discrepancy (2.1) using a learning system, namely a neural network (2.2), in a fashion similar to that of Wagner's 2021 paper (2.3). This section provides the reader with the background information needed to understand the rest of the paper.

2.1 Discrepancy theory

Discrepancy theory, sometimes called the theory of irregularities of distribution, is a subfield of combinatorics that describes the deviation of a situation from the state one would like it to be in. More informally, discrepancy theory is concerned with how much balance is possible when different elements are distributed across possibly overlapping sets.

Discrepancy in its classical form (2.1.1) has proven lower and upper bounds. Whether these bounds also apply to variants of discrepancy, such as fractional and prefix discrepancy (2.1.2), is an open conjecture (2.1.3).

2.1.1 Classical discrepancy

Although many different settings and problems exist, the classical discrepancy question goes as follows (Figure 1). Given a finite system of m sets $(S_1, \ldots, S_m)$ encompassing n elements $(1, \ldots, n)$ colored either red or blue, the discrepancy of each set is the difference (in absolute value) between the number of red elements and the number of blue elements in this set. In the example of Figure 1, namely a set system of $m = 4$ sets and $n = 8$ elements colored randomly, the discrepancy or unbalancedness is 1, 2, 0, and 0 for the four sets.

Discrepancy is not only a metric for unbalancedness in sets, but can happen at different "levels". For example, we can also compute the discrepancy of a coloring: we simply take the maximum value of the discrepancies of all sets. In the example of Figure 1, the coloring has a discrepancy of $\max\{1, 2, 0, 0\} = 2$. Coloring the elements in a different way could give us a lower coloring discrepancy. As can be imagined, there also exists an optimal coloring, which leads to the lowest coloring discrepancy possible for this particular set system. In fact, the discrepancy of a set system, yet another "level" of discrepancy, is simply the discrepancy of the optimal coloring

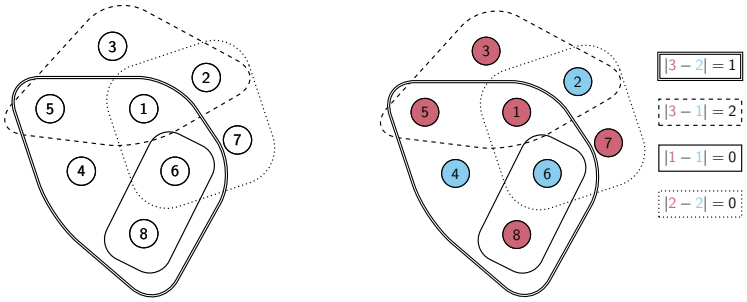

Figure 1: In this system of four sets and eight elements colored red or blue, the discrepancies of the sets are 1, 2, 0, and 0, respectively. The discrepancy of the coloring is therefore 2 (the largest of the set discrepancies).

for this system. As it turns out, the discrepancy for the set system of Figure 1 is 1, since the optimal coloring will have a discrepancy of 1 (Figure 2).

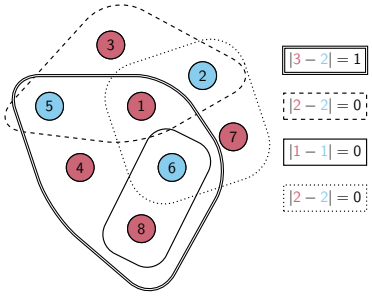

Figure 2: The optimal coloring for the set system of Figure 1, presented here, yields a discrepancy of 1 at the level of the set system.

Formally, the above set system $S_1, \ldots, S_m$ and its elements $1, \ldots, n$ can be represented by the $m \times n$ incidence matrix A, where element a_{ij} is equal to 1 if and only if element j is included in set i, otherwise it is equal to 0. Hence, the incidence matrix is to a set system what the adjacency matrix is to a graph. The coloring from Figure 1 can be represented as a vector y of size n, where blue elements get assigned a 1 and red elements a -1. Taking the absolute value of the dot product of A and y results in the same vector as on the far right of Figure 1.

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad y = \begin{bmatrix} -1 \\ +1 \\ \vdots \\ -1 \end{bmatrix} \iff |A \cdot y| = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{1}$$

Hence, to compute the discrepancy of the sets, we only need to compute $x = |A \cdot y|$, where x_i is then equal to the discrepancy of set i (Equation 1). If we want to go one discrepancy "level"

higher and find the discrepancy of the coloring, all we have to do is take the infinity norm of Ay, which gives the same result as discussed in our example:

$$\operatorname{disc}(A, y) = \|Ay\|_\infty = 2 \tag{2}$$

If we again want to go one discrepancy "level" higher and find the discrepancy of the set system, all we have to do is find the optimal coloring and compute its discrepancy, which again gives the same result as discussed in our example:

$$\operatorname{disc}(A) = \min_y \operatorname{disc}(A, y) = 1 \tag{3}$$

To summarize: we can compute the discrepancy of the sets with Equation 1, the discrepancy of the coloring with Equation 2, and the discrepancy of the set system with Equation 3. Most academic materials use the word discrepancy to mean set-system discrepancy only. We follow this convention from here on.

2.1.2 Fractional and prefix discrepancy

Whereas the previous section introduced discrepancy in its classical form, there actually exist multiple types or variants of discrepancy. These discrepancy variants are of particular interest to us, since they are lesser-known and thus still contain many open questions. In this work, we considered specifically fractional discrepancy and prefix discrepancy, two variants with interesting applications in the field of approximation algorithms.

Fractional discrepancy is essentially identical to classical discrepancy, except that the incidence matrix can contain any rational number between zero and one, instead of only the numbers zero or one, as shown in the example below.

$$A = \begin{bmatrix} 0.65 & 0.95 & 0.12 & 0.04 & 0.86 & 0.56 & 0.03 & 0.98 \\ 0.52 & 0.33 & 0.24 & 0.80 & 0.64 & 0.97 & 0.23 & 0.73 \\ 0.28 & 0.32 & 0.06 & 0.30 & 0.94 & 0.22 & 0.49 & 0.71 \\ 0.59 & 0.18 & 0.19 & 0.08 & 0.81 & 0.66 & 0.63 & 0.95 \end{bmatrix} \tag{4}$$

Prefix discrepancy also ressembles classical discrepancy, except that we add an additional maximisation step.

$$A[k] = \text{columns 0 to } k \text{ of matrix } A \tag{5}$$

$$y[k] = \text{elements 0 to } k \text{ of vector } y \tag{6}$$

$$\text{prefix-disc}(A) = \min_y \max_k \operatorname{disc}(A[k], y[k]) \tag{7}$$

Equation 7 is nearly identical to Equation 3. The main difference is that we use $A[k]$ and $y[k]$ instead of A and y, and that we pick the maximum over all k. Doing so, we actually compute the discrepancy for only the first element, the first and second elements, first to third, ..., all n elements, and then take the maximum over all these computed discrepancies to get the prefix discrepancy.

2.1.3 Open problems

Discrepancy in its classical form is relatively well-understood, with many theorems proving lower or upper bounds in particular settings. Arguably the most famous of these bounds is Spencer's theorem (Spencer, 1985), which is detailed below.

Theorem 1 (Spencer's theorem) *For any set system $\mathscr{S}$ with $n \leq m$ sets on n elements, one has $disc(\mathscr{S}) \leq \mathcal{O}\left(\sqrt{n \cdot \ln\left(\frac{2m}{n}\right)}\right)$. In particular, if $m \leq \mathcal{O}(n)$, then $disc(\mathscr{S}) \leq \mathcal{O}(\sqrt{n})$.*

Intuitively, classical discrepancy bounds should not hold for fractional and prefix discrepancy, since these variants allow for more possibilities and hence also a higher possible discrepancy. However, strangely enough, no mathematician has yet been able to find a fractional or prefix discrepancy example that did not respect these bounds. Therefore, most papers, even recent ones, still use the classical-discrepancy theorems for fractional and prefix discrepancy problems, even though this is not necessarily correct. To try and close this academic gap, we looked into whether Wagner's deep cross-entropy method can find fractional and prefix discrepancy examples that violate classical discrepancy bounds. However, given the sheer difficulty of solving long-standing conjectures, as well as the limited scope of this bachelor thesis, we focused on the following research question: can Wagner's method be adapted in such a way that it successfully finds incidence matrices with a high discrepancy (classical, fractional, and prefix)?

2.2 Neural networks

Artificial neural networks, usually simply called Neural Networks (NNs) or, more simply yet, neural nets, are a series of machine-learning algorithms used to recognize underlying relationships in a set of data. This finding of patterns in the data happens through a process similar to that of the human brain, which is where the term "neural" comes from. Although many different types of NNs exist, we only used the simplest version, called the MultiLayer Perceptron

(MLP) or, informally, the "vanilla" neural network (Figure 3). For simplicity, we are using the phrases neural network and multilayer perceptron interchangeably.

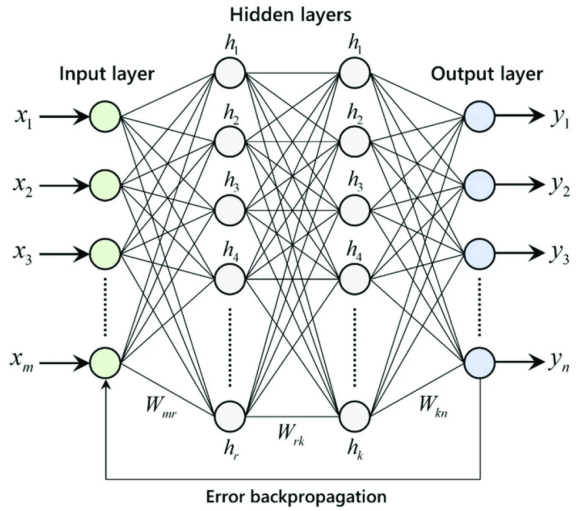

Figure 3: Besides the visible input and output layers, a multilayer perceptron includes one or more hidden layers (here, two). Each node in one layer connects with a certain weight to every node in the following layer.

In our case, we use NNs as function approximators. More specifically, a NN can be seen as a function $f(\bar{x}, W) = \bar{y}$, where the weights W get updated over time to better approximate the true function $f^*(\bar{x}) = \bar{y}^*$, as detailed in the next section (2.3). The function $f(\bar{x}, W) = \bar{y}$ transforms the m elements of input vector $\bar{x}$ into the n elements of output vector $\bar{y}$ both nonlinearly (using a so-called activation function) and linearly (using weights W) through one or more hidden layers (two in the example of Figure 3). The input elements x_i get linearly transformed (using the weights W_{nr}) into r new numbers h_i^1 in the first hidden layer. These new numbers h_i^1 then get nonlinearly transformed by an activation function, before being again linearly transformed using the weighs W_{rk} into k new numbers h_i^2 in the second hidden layer. Finally, the numbers h_i^2 get both nonlinearly transformed by an activation function and linearly transformed using the weights W_{kn} into the output layer, where they get nonlinearly transformed one last time into the output elements y_i.

2.3 Wagner's paper (2021)

As mentioned in the introduction, Wagner's paper shows how the deep cross-entropy method can be used for finding specific counterexamples to conjectures in graph theory. The reason for using this method, instead of more traditional techniques such as human reasoning or bruteforcing

with computers, is that its running time does not suffer too much from the high dimensionality of combinatorial problems. Additionally, its unique learning technique makes it more attractive than other fast algorithms (i.e., local-search algorithms).

In a nutshell, Wagner's method proceeds as follows:

1. Generate 1000 random graphs using a probability distribution $f(\bar{x}, W)$,
   where $\bar{x}$ is the graph and W is a set of weights.

2. Give a "score" to each graph, rank the graphs by score,
   and select the top 10% of graphs according to this score.

3. Update the weights w so that the distribution $f(\bar{x}, W)$
   is more likely to generate this top 10% of graphs.

4. Repeat (until you find a counterexample).

Step 1 To generate a random graph on n vertices, all one has to do is generate a random vector of size $\frac{n(n-1)}{2}$, containing only zeros and ones. This random vector can then be reshaped into an adjacency matrix, which is enough to represent a graph. However, it should not be generated randomly (uniform distribution), but rather according to a specific and updatable distribution function $f(\bar{x}, W)$.

We use a NN as a Probability Distribution Function (PDF) to generate the random graphs of n vertices, or equivalently, the zero-one vectors of size $\frac{n(n-1)}{2}$, as follows (Figure 4). We first ask the network to predict what the best first vector element should be. The output is a probability distribution on the elements zero and one, from which we can sample an element randomly and feed it back into the network to ask what the best second vector element should be (a zero or a one). In general, having generated the elements $a_1, a_2, \ldots, a_{k-1}$, we can feed this partial vector on $k-1$ elements into the network to get a probability distribution on the best next element to use, conditioned on our previous $k-1$ decisions, and sample from it randomly to obtain a_k. Repeating this process $\frac{n(n-1)}{2}$ times, we get a random graph generated by the probability distribution $f(\bar{x}, W)$.

The problem with the above approach is that the network cannot assess which element it is currently predicting. Imagine the input vector is $\bar{x} = [011000]$, does this mean that edges 4 and 5 have already been considered and were given a value of zero, or haven't been considered yet and thus still have a value of zero? To solve this problem, we add another vector of size $\frac{n(n-1)}{2}$

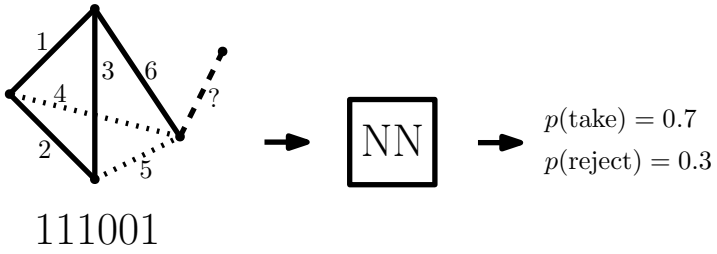

Figure 4: We are generating a graph, and so far the network has included edges 1,2,3,6, and rejected edges 4,5 (dotted), corresponding to the sequence 111001. We input this into the neural network to get a probability distribution on whether to include or reject the next edge.

as input. This second vector will be filled with zeros, except on one spot where the vector element will be equal to one. In this way, the network knows which edge it currently has to predict (the element with a one) and which edges have already been considered (all elements before this nonzero element). For example, if the input vector is now $\bar{x} = [011000] + [000010]$, we know that edge 4 has already been considered and given value zero (it is before the nonzero element), and that edge 5 is currently being considered (it has a one in the second input vector).

Step 2 Once the 1000 random graphs have been generated, one needs to give them a score. This score is essentially a guideline to show the artificial-intelligence agent in which direction it should go. As such, the most intuitive way to define the scoring function is to base it on the conjecture we want to disprove. For example, imagine one would like to disprove the following trivial conjecture: graphs with 20 vertices cannot have more than 50 edges. In this case, an intuitive scoring function would be the number of edges in the graph, meaning a graph with 30 edges gets a score of 30. By defining the scoring function in this way, we make sure the distribution $f(\bar{x}, W)$ will be updated to have a higher chance of generating graphs with a lot of edges (thus a high score), which is exactly what we want. If we repeat this process enough times, we should stumble on a counterexample to our trivial conjecture.

Step 3 Once we have generated 1000 graphs and ranked them by score, we only need to update the weights in order to finish the iteration.

Although using NNs to generate random graphs works fine, there exist plenty of easier methods. However, the beauty of using NNs in this setup is not the generating of the graphs, but rather the learning, over time, of a better probability distribution. Using cross-entropy, gradient descent, and backpropagation, the network easily and automatically updates its weights a little bit to make it more likely to output moves that were used in the graphs with the highest

scores. Essentially, the model reinforces the generation of graphs with high scores, hoping to eventually generate a counterexample to the conjecture. This updating of the weights over time is equivalent to saying the network continuously approximates the probability density function that generated the top 10% of graphs, hence the name "function approximator". Detailing how this approximation (the updating of the weights) works is out of the scope of this paper.

3 Methodology

To find matrices with a high discrepancy, we first need to implement and adapt Wagner's code to discrepancy theory, essentially ensuring that the basic setup works (3.1). Afterwards, we need to make sure that the scoring function is well-defined, so that the model can receive enough signal (3.2) to distinguish desirable from less desirable incidence matrices. Lastly, we don't want to wait for weeks before obtaining interesting results, which is why we also optimize for speed (3.3).

3.1 Basic setup

Although Wagner uses the deep cross-entropy method for finding counterexamples in graph theory, his code was written in such a way that it could easily be adapted to other subfields of combinatorics. On top of that, the code is easily accessible on GitHub[1] and free to use. Therefore, most of our code was built upon his, and most of our setup decisions are thus his.

Wagner's code (and therefore also ours) is written in Python, most probably due to the simplicity of the language as well as the abundance of great machine-learning and neural-network packages available. The code can run on most laptops available today and is written in a way that's easy to understand. We placed our adaptation of his code on GitHub as well[2].

To adapt the code to prefix discrepancy, we only needed to change the `calc_score` function, namely the function that takes an instance as input and returns the instance's score. Since we want to find high (prefix) discrepancy matrices, we defined our scoring function (`calc_score`) to take an incidence matrix as input and to return the matrix's prefix discrepancy (Equation 7) as score. In this way, the network should, over time, learn to find matrices of very high prefix discrepancy. If this occurs, we will have succeeded in showing that Wagner's method can successfully be applied to discrepancy theory, and hence maybe also be used to disprove some of the conjectured upper bounds.

For fractional discrepancy, adapting Wagner's code was slightly trickier, as the NN now needs to generate continuous numbers instead of only zeros and ones (see Section 3.2). The scoring function (`calc_score`) is defined in the same way as for prefix discrepancy, except it now returns the fractional discrepancy (Equation 3) as score.

[1] `https://github.com/zawagner22/cross-entropy-for-combinatorics`

[2] `https://github.com/supercomputer-lars/thesis`

3.2 Issue #1: signal

In order to learn, the neural network needs to be able to differentiate desirable from less desirable matrices on the basis of their respectives scores—what we call the "signal". Unfortunately, we encountered signal issues for our implementation of both prefix discrepancy (3.2.1) and fractional discrepancy (3.2.2).

3.2.1 Prefix discrepancy

The first problem we encountered after adapting Wagner's code for prefix discrepancy is that the neural network was not receiving enough signal: we were not giving the NN enough information about which matrices were desirable (meaning the NN should learn from them) and which matrices were less desirable (meaning the NN should not learn from them). Just like classical discrepancy, prefix discrepancy is a discrete number, and tends to be the same for matrices of the same size. For example, most 7×7 matrices have a prefix discrepancy of 2. Therefore, if they all get a score (= prefix discrepancy) of 2, the NN will not know which ones are desirable and which ones are less so. The step-by-step example below illustrates this problem, as it follows the steps of the deep cross-entropy method.

1. We use the NN to generate 1000 random incidence matrices.
2. We give all matrices a score, namely their prefix discrepancy, and rank the matrices by score. However, since all matrices will almost surely have a prefix discrepancy of 2, the ranking will not change the order of the matrices.
3. We update the weights of the NN so that it's more likely to generate matrices with a high prefix discrepancy. However, since the ranking did not really do anything, the model will not learn from the top 10%, but rather from the first 10% of the sample, which is really a random sampling.
4. Repeating these steps over and over will not help the model find an incidence matrix of high prefix discrepancy. This is because our approach is not better than randomly generating matrices (from a uniform distribution), since the NN "learns" from random examples only.

As a solution to the above problem, we had the model select, among the many matrices with the same prefix discrepancy, those with the fewest colorings yielding this discrepancy. To this end, we modified the scoring function to penalize matrices having many colorings yielding the

prefix discrepancy, as follows:

$$\texttt{calc\_score}(A) = \text{prefix-disc}(A) - 0.0001 \cdot \ln(\texttt{count}) \tag{8}$$

where `count` is a variable containing the number of colorings that lead to the prefix discrepancy. Now, instead of learning from essentially a random 10% of the matrices, the model learns from the 10% with the fewest colorings leading to a prefix discrepancy of 2. For example, it will learn from models with score 1.99 more than from models with score 1.97. If we repeat this process enough times, the model will find a matrix with only one coloring that leads to a prefix discrepancy of 2. In our example, this would be equal to a score of 2.00, since $\ln(1) = 0$. From here on, the model can easily find a matrix with zero colorings that lead to a prefix discrepancy of 2, meaning it has found a matrix with a prefix discrepancy of 3. Afterwards, the same process repeats, until the model is unable to "jump" to a higher prefix discrepancy (either because it does not exist or because it's very hard to find).

3.2.2 Fractional discrepancy

As mentioned in Section 2.3, we can generate random matrices by having a NN output a PDF over the elements zero and one. In practice, this is done by using a sigmoid activation function on the output neuron, leading the NN to output a continuous number y between zero and one. From here on, we can use y as a PDF over zero and one, with y being the probability of generating a one and $(1 - y)$ being the probability of generating a zero.

To adapt Wagner's code to fractional discrepancy, we need to modify the NN so that it can generate fractional matrices. A naive way to do so would be to simply let y represent the fractional value. For example, if we are generating the third element of a fractional matrix and the NN outputs $y = 0.389$, then we simply fill the third element in the matrix by 0.389. This approach, however, removes any randomness: the NN will now generate a 1000 identical fractional matrices, which will lead us to a signaling problem very similar to the one for prefix discrepancy (Section 3.2.1). In this case, however, adding a `count` variable will not change anything, as all matrices are perfectly identical and thus have the exact same value for the `count` variable.

To solve the problem, one could add white noise ϵ to y, giving us $y^* = y + \epsilon$, with $\epsilon \sim N(0, \sigma^2)$. This white noise must be small enough so the network can still choose which matrix it wants to generate, but large enough so the 1000 generated matrices are still different from

each other, making sure the network receives enough signal about which matrix structures are more desirable than others (essentially enough examples of more and less desirable matrices). Although this approach seemed to work well for a few iterations, the network was quickly stagnating afterwards. Most probably, the network is not able to express a so-called "confidence level". In the traditional binary case, the model can output a y very close to zero, meaning the network is highly confident that this matrix element should be a zero, since the probability of generating a zero, expressed by $1 - y$, is now very high. However, if the model is not very confident about whether the following matrix element should be a one or a zero, it will simply output a value around 0.5, meaning there is now an equal chance for the next element to be zero or one, as desired. In the fractional case, in contrast, if the NN is highly confident about which value the next matrix element should have, it cannot express this confidence, since ϵ will add too much random noise to the final value. For example, if the NN is sure that the next matrix element should be 0.8, it will output 0.8, but since we add ϵ, it could very well be that the final value turns out to be 0.7 instead of 0.8, suggesting we need a smaller ϵ. Similarly, if the fractional NN is unsure about the next element, it cannot express this uncertainty, since outputting 0.5 means the final value will still be very close to 0.5 (e.g., 0.4 or 0.6), whereas we would have liked a random number from all possible values between zero and one; in this case, we actually need a larger ϵ. In other words, we had to dig deeper for a solution.

After multiple other attempts, we came up with the following elegant solution, which, interestingly enough, is derived from techniques used in image recognition. Consider a NN that does not output only one number y, but rather 11 numbers y_j $(j = 0 \dots 10)$, each representing a value from the sequence 0.0, 0.1, ..., 0.9, 1.0. In this scenario, if the NN is highly confident that the next matrix element should be, say, 0.3, then it will simply output a number close to one for y_3 and numbers close to zero for the other y_j $(j \neq 3)$. However, if the NN is highly unsure about what the next fractional element should be, it can simply output the same value for all y_j, meaning all numbers in the sequence now have the same probability of being sampled. Essentially, this new NN structure generates a PDF, not over the numbers zero and one, but over all 11 numbers in the sequence. Afterwards, the NN will automatically adjust its weights to better approximate the distributions that generated the top 10% of matrices. We could have outputted more than 11 numbers, meaning the network would be able to choose from more precise numbers, for example 0.35. Basically, the larger the number of output values you choose, the more numbers after the digits the network will be able to generate. However, it is best to keep this amount relatively small, otherwise the output layer is disproportionately large compared to the other layers of the network, which is not recommended in standard NN theory.

3.3 Issue #2: speed

When adapting Wagner's code to prefix and fractional discrepancy, we encountered a second issue: speed. Speed plays a crucial part in the deep cross-entropy method, since the method needs to run for possibly many iterations before it can find a counterexample. At first, this might seems counterintuitive, seeing that the NN learns over time and thus should find the counterexample much faster than a bruteforce approach. However, consider a combinatorial setting where bruteforcing a counterexample takes, on average, around a trillion iterations. In this case, even if our deep cross-entropy method can find the counterexample, on average, around a million times faster, it's still going to need a million iterations. Therefore, making sure that every iteration goes as fast as possible is crucial for finding a counterexample in a reasonable amount of time. To do so, we optimized the speed of our program at two different levels: the machine level (3.3.1) and the algorithms level (3.3.2).

3.3.1 Machine-level optimization

We successfully increased the speed of our code at the machine level in three different ways.

The first speed improvement was suggested by Wagner's code, which uses the package Numba, a just-in-time compiler for Python that works best on code using NumPy arrays and functions, as well as loops. Since our own code uses many NumPy arrays and functions (e.g., dot product of incidence matrix and coloring vector) as well as many loops (e.g., trying out all colorings to find the optimal one), this package is of great help to us. Although slightly tricky to work with, it is very effective at improving the performance of interpreted programs such as Python.

The second speed improvement we made is very simple, yet also very effective and strangely enough not implemented by Wagner. It works, however, with prefix discrepancy only (and not with fractional discrepancy). Whenever we create a NumPy array, we make sure to define it as having a data type of `np.int8`, so the elements in the NumPy arrays are represented by a single byte, instead of four or eight as in `np.int32` and `np.int64`. Since all elements in our arrays have value zero or one, this change will not affect the validity of our code. What it will do, however, is significantly reduce the size of the arrays and hence also speed up the computations.

As a final major machine-level optimization, we parallelized our code. Instead of continuously generating and scoring 1000 random samples using one CPU, we asked all eight CPUs in our

laptop to each generate and score 125 samples, meaning we cut our computation time by a factor 8. Since the samples are independent and identically distributed, we can indeed generate and score them separately, without one CPU having to wait for information from another one. To make this parallelization even more powerful, we used Google's Cloud Compute Engine to run our code on a virtual machine. In this way, we can run our code on virtual machines with 16 or 32 CPUs, effectively cutting our computational time by a factor 16 or 32 (or even more if we wish to). This parallelization and use of cloud resources was our own idea and effort; Wagner did not mention or use these methods in his paper or in the corresponding code.

3.3.2 Algorithmic optimization

Computing classical, prefix, or fractional discrepancy requires iterating over all colorings and finding the optimal one (the one with lowest discrepancy). Since there are 2^n possible colorings for an $m \times n$ incidence matrix, computing discrepancy is NP-hard, meaning we cannot solve it in polynomial time. What we can do, however, is realize that opposite colorings, such as vectors $\bar{y}_1 = [+1\ -1\ +1]$ and $\bar{y}_2 = [-1\ +1\ -1]$, have the same discrepancy: red elements have simply been replaced by blue ones and vice versa. Accordingly, we could reduce the amount of iterations by a factor 2, by iterating over the colorings that start with 1 only and ignoring those that start with -1.

Although the focus of this bachelor thesis was only to find matrices of high discrepancy (using Wagner's method), we did so with in mind the broader goal of solving open questions in discrepancy theory (mostly regarding upper bounds). One of these open questions goes as follows: given a set system of m sets containing n elements and with $m \ll n$, what is the upper bound for the prefix discrepancy of the set system? For the opposite setting, namely $n \leq \mathcal{O}(m)$, prefix discrepancy is asymptotically the same as classical discrepancy up to constants (Spencer, 1986). Therefore, to close the academic gap, we wanted the NN to try and find $m \ll n$ matrices with a high prefix discrepancy, in the hope that this will tell us more about upper bounds in the only unknown setting ($m \ll n$). Consequently, we developed a dynamic program that is efficient at computing the prefix discrepancy of a set system with few sets and many elements. In this way, we can work with matrices where n is really large, without being limited by speed issues. The idea behind the dynamic program is to construct a table $D(i, v)$ of size $n \times (2d+1)^m$, which can then be used to determine whether the incidence matrix has a prefix discrepancy of less than or equal to a constant d. The exact construction of this table $D(i, v)$ is detailed below. Once we

know how to construct the table, we can use it to find the prefix discrepancy by augmenting d until the table outputs TRUE, meaning that the discrepancy is then equal to this last d. Using this dynamic program, we can compute the prefix discrepancy in $\mathcal{O}(n \cdot (2d+1)^m)$, which is linear in the number of elements and exponential in the number of sets. Therefore, if we keep m small enough, we can still relatively quickly compute the prefix discrepancy for very large n. This dynamic program can fairly easily be adapted to also output the amount of colorings that lead to the prefix discrepancy (`count` variable), meaning we can also use it to compute Equation 7.

Define $D(i, v)$ as follows:

$$D(i,v) = \begin{cases} \text{true}, & \text{if we can color elements } 1, 2, \ldots, i \text{ in a way that the discrepancy} \\ & \text{of all prefixes until column } i \text{ is at most } d \textbf{ and} \text{ the signed sum} \\ & \text{(summing all columns multiplied by their color) equals } v \\ \text{false}, & \text{otherwise} \end{cases}$$

Initialization:

$$D(0,v) = \begin{cases} \text{true}, & \text{if } v \text{ is the zero vector} \\ \text{false}, & \text{otherwise} \end{cases}$$

Iteration:

$$D(i+1,v) = \begin{cases} \text{true}, & \text{if } v - A(i+1) \text{ is bounded by } d \text{ in all coordinates} \\ & \textbf{and } D(i, v - A(i+1)) = \text{true} \\ \text{true}, & \text{if } v + A(i+1) \text{ is bounded by } d \text{ in all coordinates} \\ & \textbf{and } D(i, v + A(i+1)) = \text{true} \\ \text{false}, & \text{otherwise} \end{cases}$$

Output:

$$\text{prefix-disc}(A) \leq d = \begin{cases} \text{true}, & \text{if } D(n,v) = \text{true for any } v \\ \text{false}, & \text{otherwise} \end{cases}$$

4 Findings

We implemented the techniques described in Section 3 and evaluated whether they are successful at finding high discrepancy—again for all three variants: classical (4.1), prefix (4.2), and fractional (4.3). Initially, we worked with the same NN as Wagner, who used three hidden layers of respectively 128, 64, and 4 neurons. For prefix and fractional discrepancies, and in an attempt to improve the learning, we eventually quadrupled the number of neurons in each layer.

4.1 Classical discrepancy

Although our final goal is to find counterexamples for prefix and fractional discrepancy, we first wanted to verify whether our NN is capable of reaching the upper bounds proven for classical discrepancy. Practically, 7×7 matrices have an upper bound of discrepancy 3, meaning one cannot construct a zero-one matrix of size 7×7 that will have a (classical) discrepancy higher than 3 (Rohwedder, 2022). Most 7×7 matrices, however, have a discrepancy of 2. Therefore, we wanted to see whether our network was able to find one of the few 7×7 matrices with a classical discrepancy of 3, essentially reaching the upper bound.

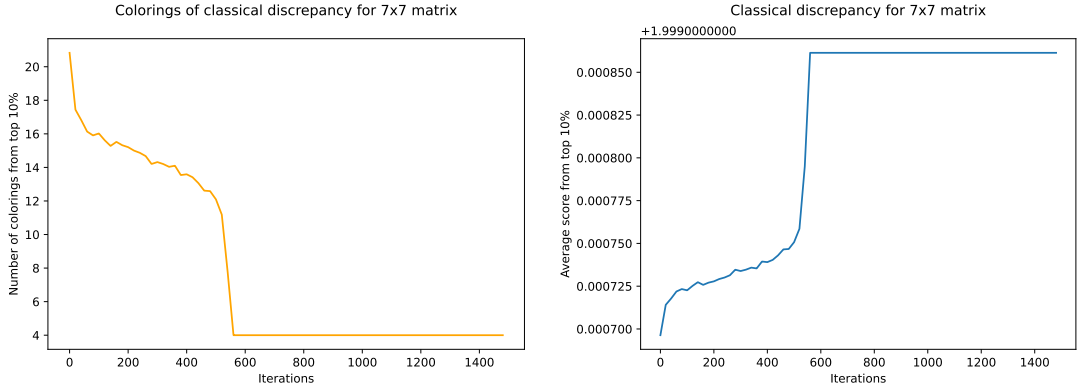

Figure 5: In about 500 iterations, our NN learned to go from about 20 to just 4 colorings for matrices with a classical discrepancy of 2 (left), corresponding to scores according to Equation 8 of about 1.999700 to 1.999861, respectively (right). Average data for the top 10% (top 100 matrices) every 20 iterations.

Our NN starts off well (Figure 5), by continuously finding matrices with a better score, calculated according to Equation 8. All these matrices have a classical discrepancy of 2, but progressively fewer colorings. However, instead of increasing all the way to 3, the score increases sharply and starts to plateau around 500 iterations at a value corresponding to four colorings, for unclear

reasons. Nevertheless, this is still an encouraging result: without prior understanding of the problem setting, our NN is able to automatically find patterns in the data so as to maximise the given score function (Equation 8). Given that there are extremely few 7×7 matrices with a classical discrepancy of 3, finding even a single one is a challenging task. Although it did not reach the proven upper bound, the model is clearly capable of learning and hence could still possibly find counterexamples to less difficult problems. Perhaps the parameters we used were not optimal. For example, changing the learning rate might lead the NN to find these few matrices with a discrepancy of 3. As a matter of fact, even well-designed NN models often get stuck in local optima because of an inadequate learning rate. Consequently, we believe the experiment is worth repeating with different learning rates (or other influential parameters). For this thesis, however, we had enough evidence to move on to the discrepancy variants.

4.2 Prefix discrepancy

We repeated the experiment, this time trying to find an incidence matrix with a high prefix discrepancy instead of a high classical discrepancy. As it turned out, our model is extremely good at finding matrices with a prefix discrepancy of 3 (Figure 6, left): the average score of the top 10% was nearly 3 after only 30 iterations (not exactly 3 since it is again computed with Equation 8, taking the number of colorings into account). Because the NN generates 1000 random matrices at every iteration, the network only generated around 30 000 random matrices before finding over 100 matrices with a prefix discrepancy of 3. As the deep cross-entropy method is a random process, we repeated the experiment 100 times and got an average of 21 540 random matrices before finding one with a prefix discrepancy of 3. In contrast, when we sampled from a uniform distribution, we needed over three million random matrices (on average over 100 trials) before finding a matrix with a prefix discrepancy of 3 (Table 1).

These early results are very encouraging: clearly, the NN is able to continuously learn over time, without any prior knowledge of the problem. The most impressive learning takes place around 23 iterations (Figure 6, left): after having found one matrix with a prefix discrepancy of 3, the NN can very quickly generate more than 100 other matrices with the same discrepancy, leading to a sharp rise in the average score of the top 10%. Also, introducing the number of colorings (`count` variable) in the scoring function (`calc_score`) worked: the NN is continuously improving the score of the top 10%, meaning the network is continuously generating matrices with fewer colorings from the very first iteration (Figure 6, right). Then, at some point, the

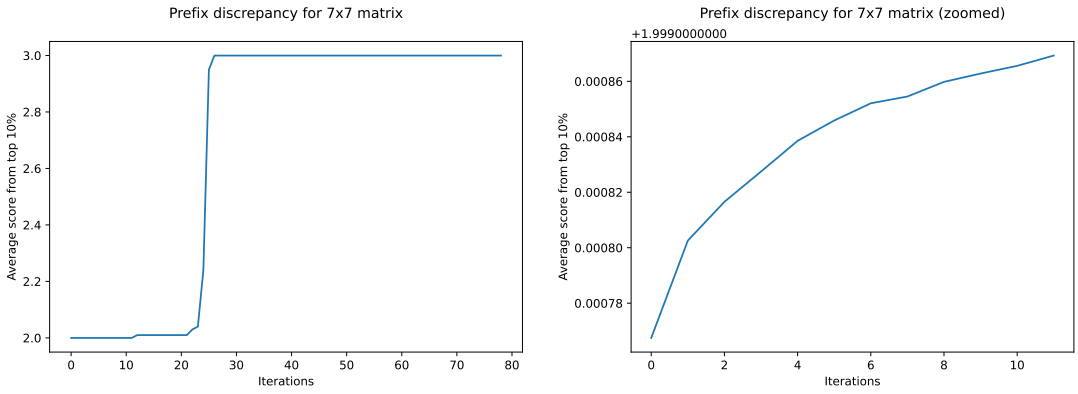

Figure 6: Our model did better on prefix discrepancy than on classical discrepancy: in only 30 iterations, it could generate 10% of matrices with a prefix discrepancy of 3 (left). The introduction of the number of colorings in the scoring function did solve the signal issue: even when most or all matrices still have a prefix discrepancy of 2, the NN is able to learn at every iteration from the very first one (right). Average data for the top 10% (top 100 matrices) at every iteration.

Table 1: For finding a single matrix with a prefix discrepancy of 3, our NN approach was almost 150 times more efficient than a naive approach. For finding a hundred such matrices, our NN approach was over ten thousand times more efficient. The results shown here are averages over 100 trials.

Task	**Method**	**# matrices**
Find 1 matrix with pref-disc. 3	naive	3 141 584
	NN	21 540
Find 100 matrices with pref-disc. 3	naive	314 158 400
	NN	28 510

NN is able to make the jump from few colorings that lead to a prefix discrepancy of 2 to zero coloring with a prefix discrepancy of 2, meaning it has found a matrix with a prefix discrepancy of 3. Why this jump is possible for prefix discrepancy and not for classical discrepancy is not really clear. Perhaps there are more matrices with a prefix discrepancy of 3, confirming that our model does work perfectly when the task is slightly less difficult than the previous experiment. Or perhaps prefix discrepancy is less sensitive to the learning rate than classical discrepancy.

The next step is to replace the 7×7 matrices by $m \times n$ matrices with $m \ll n$, a more relevant case for our final goal. The few preliminary experiments we had time to conduct were not conclusive, but our code is ready for interested readers (with more time on their hands) to conduct their own experiments and thus further explore the model's potential.

4.3 Fractional discrepancy

As a last experiment, we evaluated the network's potential at finding 7×7 matrices with a high fractional discrepancy. This time, however, we did not use Wagner's proposed NN model. Instead, we used our newly developed model for fractional combinatorics, presented in Section 3.2.2. Moreover, we did not use Equation 8 (with `count`) as scoring function, since fractional discrepancy does not have the issue of having to "jump" from one discrete discrepancy to the next discrete discrepancy, for example from a discrepancy of 2 to a discrepancy of 3.

Our new NN model was able to continuously learn and improve the average top 10% score for the first 60 000 iterations (Figure 7), after which it seemed to stagnate around a top 10% average of 2.00. This result is already encouraging: the new model we developed successfully learned how to find matrices with fractional values that lead to a high discrepancy.

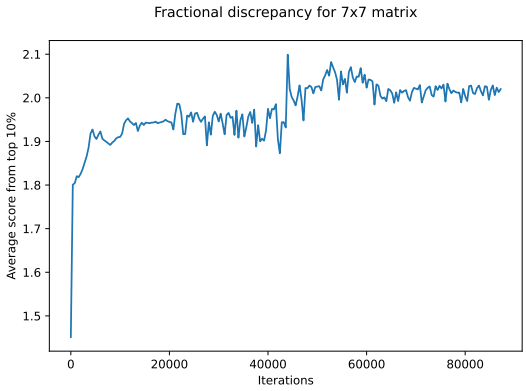

Figure 7: Although it needed as many as 60 000 iterations, our new model was able to find matrices with a fractional discrepancy greater than 2 (which indirectly led to nonfractional matrices with a classical discrepancy of 3—the upper bound for 7×7 matrices). Average data for the top 10% (top 100 matrices) every 20 iterations.

As it turns out, the result we obtained is even better than expected: instead of hovering around an average of 2.00, the score actually hovers around an average slightly above 2.00, meaning the network is consistently able to find at least a few matrices with a fractional discrepancy higher than 2. Interestingly, when we replaced all fractional values in these matrices by their closest integer (zero or one), we got binary matrices with a classical discrepancy of 3. Indirectly, our model was thus able to find matrices with the highest classical discrepancy possible. In other words, our new model allowed us to reinforce desirable behavior with a more discriminating signal, leading after discretization to these extremely hard-to-find binary matrices. Accordingly, our fractional model could be seen as a potential relaxation technique for the binary problem.

Still, whether this property also holds for matrices of size other than 7×7 remains to be seen. Also, changing some of the input parameters might lead to a different result, perhaps even a better result, such as finding these binary matrices directly (without need for discretization). To answer these relevant questions, we believe more experiments should be run.

5 Conclusion

After reading the encouraging results described in Wagner's preprint (2021), we wondered whether we could use his technique, namely the deep cross-entropy method, to solve open problems in discrepancy theory, instead of graph theory. More specifically, we focused on finding matrices with a high discrepancy (classical, prefix, and fractional), since this is the first step to disproving conjectured upper bounds. To do so, we increased the overall speed of the program by performing machine-level and algorithmic optimizations, as well as increased the method's signal generation, among others by developing a new NN model for fractional combinatorics. For the few experiments we ran, the results are very promising: the NN is able to effectively find examples of higher discrepancy for all three variants. Additionally, our newly developed fractional model turned out to excel at generating more signal, so much so that the model even outperforms Wagner's method in binary settings. This thesis thus represents a proof-of-concept for adapting Wagner's method to finding examples of high discrepancy.

Given these encouraging results, we believe further research is to be pursued. As a first step, researchers should run more experiments, to see how the model performs in different settings: different matrix sizes (e.g., $m \ll n$), different NN parameters (e.g., learning rate and number of neurons), higher dimensions (e.g., 1000×1000), etc. During this process, more issues and interesting discoveries will likely emerge, paving the path for further developments. Some of these further developments could come from very different fields. In the same way that image-recognition techniques inspired our fractional model, we believe other fields, such as chemistry (De Cao & Kipf, 2018), might contain interesting ideas for the task at hand. Further research need not be restricted to discrepancy theory. For example, researchers could use our new NN model to find counterexamples to open conjectures in fractional combinatorics subfields.

References

De Cao, N., & Kipf, T. (2018). *MolGAN: An implicit generative model for small molecular graphs* (No. arXiv:1805.11973). Retrieved 2022-06-30, from http://arxiv.org/abs/1805.11973 (type: article) doi: 10.48550/arXiv.1805.11973

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2022). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. Retrieved 2022-06-22, from https://www.nature.com/articles/nature14236 (Number: 7540 Publisher: Nature Publishing Group) doi: 10.1038/nature14236

Pan, X., You, Y., Wang, Z., & Lu, C. (2017). *Virtual to real reinforcement learning for autonomous driving* (No. arXiv:1704.03952). Retrieved 2022-06-22, from http://arxiv.org/abs/1704.03952 (type: article)

Rohwedder, L. (2022). *Private communication.*

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. Retrieved 2022-06-22, from https://www.nature.com/articles/nature24270 (Number: 7676 Publisher: Nature Publishing Group) doi: 10.1038/nature24270

Spencer, J. (1985). Six standard deviations suffice. *Transactions of the American Mathematical Society*, *289*(2), 679–706. Retrieved 2022-06-23, from https://www.ams.org/tran/1985-289-02/S0002-9947-1985-0784009-0/ doi: 10.1090/S0002-9947-1985-0784009-0

Spencer, J. (1986). Balancing vectors in the max norm. *Combinatorica*, *6*(1), 55–65. Retrieved 2022-06-28, from https://doi.org/10.1007/BF02579409 doi: 10.1007/BF02579409

Zhang, Z., Zohren, S., & Stephen, R. (2020). Deep reinforcement learning for trading. *The Journal of Financial Data Science*. Retrieved from https://jfds.pm-research.com/content/early/2020/03/16/jfds.2020.1.030 doi: 10.3905/jfds.2020.1.030