THE INFLUENCE OF COMBINATORIAL DESIGNS USED IN DEEP NEURAL NETWORKS TO PREVENT OVERFITTING

(DERİN SİNİR AĞLARINDA AŞIRI ÖĞRENMEYİ ENGELLEMEK İÇİN KULLANILAN KOMBİNATÖRYEL TASARIMLARIN ETKİSİ)

by

MUHAMMET ALİ ÖZTÜRK, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER

in

MATHEMATICS

in the

GRADUATE SCHOOL OF MATHEMATICS

of

GALATASARAY UNIVERSITY

JUNE 2021

Approval of the thesis:

THE INFLUENCE OF COMBINATORIAL DESIGNS USED IN DEEP NEURAL NETWORKS TO PREVENT OVERFITTING

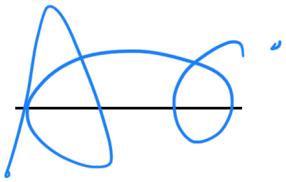
submitted by **MUHAMMET ALİ ÖZTÜRK** in partial fulfillment of the requirements for the degree of **Master in Mathematics Department, Galatasaray University** by,

Examining Committee Members:

Prof. Dr. A. Muhammed Uludag
Supervisor, **Mathematics Department, Galatasaray University**

Prof. Dr. Sibel Özkan
Mathematics Department, Gebze Technical University

Assist. Prof. Dr. Ayşegül Yıldız Ulus
Mathematics Department, Galatasaray University



-

-

Date:

ACKNOWLEDGMENTS

I thank a lot to everyone who helped me while writing this thesis. I also thank a lot my mother, my father and my wife because of their immense supports in general.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Combinatorial designs, as a mathematical subject, has deep connections with the fields such as combinatorics, graph theory, finite geometry, coding theory, cryptography and number theory and has attracted many researches from different fields.

The combinatorial design found in the book Brhat Samhita, written by Varahamihira around 587 BC, was used to produce perfume by selecting 4 items from 16 different items with the help of a magic square. This is first known example of combinatorial designs. Combinatorial designs are used in widely known Kirkman's school girl problem. In the widely accepted view, modern study of block designs began in 1936 with the publication of an article by the statistician F. Yates.

In this thesis, firstly, we provide an introduction about combinatorial designs and their properties. We will also introduce machine learning fundamentals that is required for our problem. Then, the problem that is being solved to generate BIBDs, exact covering problem will be mentioned. This NP-complete problem can be solved using Donald Knuth's Algorithm X. At this step, we will be explaining Algorithm X's algorithm and its implementation. Generated BIBDs will be used in neural networks, which have been built manually to prevent overfitting in machine learning.

We are showing that it is possible to prevent overfitting in machine learning succesfully by using a known pattern (BIBDs) in dropout. And finally, we will share our results and compare them with widely known and used random dropout method. We will also compare our results with no-regularization applied models.

Keywords : Combinatorial designs, neural networks, machine learning, overfitting, dropout, deep learning

RÉSUMÉ

Les désigns combinatoires, en tant que sujet mathématique, ont des liens profonds avec des domaines tels que la combinatoire, la théorie des graphes, la géométrie finie, la théorie du codage, la cryptographie et la théorie des nombres et ont attiré de nombreuses recherches dans différents domaines.

Le premier exemple connu de conception combinatoire se trouve en Inde dans le livre Brhat Samhita de Varahamihira, écrit vers 587 après JC, dans le but de faire des parfums en utilisant 4 matières sélectionnées parmi 16 matières différentes en utilisant un carré magique. Les conceptions combinatoires sont également utilisées dans le problème largement connu des écolières de Kirkman. Selon l'opinion largement acceptée, l'étude moderne des conceptions de blocs a commencé avec la publication en 1936 d'un article du statisticien F. Yates.

Dans cette thèse, tout d'abord, nous fournissons une introduction sur les conceptions combinatoires et leurs propriétés. Nous présenterons également les principes de base de l'apprentissage automatique nécessaires à notre problème. Ensuite, le problème qui est résolu pour générer des BIBD, le problème de couverture exact sera mentionné. Ce problème NP-complet peut être résolu en utilisant l'algorithme X de Donald Knuth. À cette étape, nous expliquerons l'algorithme de l'algorithme X et son implémentation. Les BIBD générés seront utilisés dans les réseaux de neurones construits manuellement pour éviter le surajustement dans l'apprentissage automatique.

Nous montrons qu'il est possible avec succès d'éviter le surapprentissage en apprentissage automatique en utilisant un modèle connu (BIBD) au lieu de l'aléatoire dans le décrochage. Enfin, nous partagerons nos résultats et les comparerons avec une méthode d'abandon aléatoire largement connue et utilisée. Nous comparerons également nos résultats avec des modèles appliqués sans régularisation.

Mots Clés : Les designs combinatoires, réseaux de neurones, l'apprentissage automatique, le surajustement, dropout, l'apprentissage en profondeur

ÖZET

Bir matematik konusu olarak kombinatoryal tasarımlar, kombinatorik, grafik teorisi, sonlu geometri, kodlama teorisi, kriptografi ve sayı teorisi gibi alanlarla derin bağlantılara sahiptir ve farklı alanlardan pek çok araştırmanın ilgisini çekmiştir.

Bilinen ilk kombinatoryal tasarım örneği Hindistan'da, sihirli bir kare kullanarak 16 farklı maddeden seçilen 4 maddeyi kullanarak parfüm yapmak amacıyla MS 587 civarında yazılan, Varahamihira'nın Brhat Samhita kitabında bulunur. Kombinatoryal tasarımlar, yaygın olarak bilinen Kirkman'ın okul kızı probleminde de kullanılmaktadır. Yaygın kabul gören görüşe göre, modern blok tasarım çalışmaları, 1936'da istatistikçi F. Yates'in yazdığı bir makalenin yayınlanmasıyla başlamıştır.

Tezimize kombinatoryal tasarımları anlatarak giriş yapacağız. Sonrasında ise çözmek istediğimiz problemin türünden, tam kaplama probleminden, bahsedeceğiz. Bu problem Donald Knuth'un X Algoritmasını kullanarak çözülebiliyor. Dolayısıyla, daha sonrasında aşırı öğrenmeyi sinir ağlarında önlemek için kullanacağımız dengeli tamamlanmamış blok tasarımları tam kaplama problemini çözerek üretecek olan algoritmadan ve aşamalarından bahsedeceğiz.

Bu tezde, rastlantısallık yerine bilinen bir örüntüyü (BIBD'leri) kullanarak makine öğreniminde aşırı uyumu başarıyla önlemenin mümkün olduğunu gösteriyoruz. Son olarak, sonuçlarımızı paylaşacağız ve bunları yaygın olarak bilinen ve kullanılan rastgele bırakma yöntemiyle karşılaştıracağız. Ayrıca sonuçlarımızı, hiçbir düzenleme uygulanmayan modellerle de karşılaştıracağız.

Anahtar Kelimeler : Kombinatoryel tasarımlar, yapay sinir ağları, makine öğrenmesi, aşırı öğrenme, dropout, derin öğrenme

1 LITERATURE REVIEW

In this thesis, we will mostly be focusing on combinatorial objects "balanced incomplete block designs" and artificial "neural networks". We will start from balanced incomplete block designs side and try constructing a bridge between. Starting point of this research was the desire to use existing combinatorial objects that we generated using Donald Knuth's Algorithm X in some way and the beleif we had in having better results using dropout designs in test datasets than we do have with usual dropout methods. This would provide less randomness and more explicability to our neural networks. The method we will introduce can be applied to deep neural networks but in this thesis, we kept neural networks small that is with maximum 7 layers.

We used various number of combinatorial design books while learning combinatorial designs and its properties. I have taken and completed succesfully Design Theory 1 PhD course from Yıldız Technical University from Fatih Demirkale for this purpose. Then we made a deep research about combinatorial designs' applied areas and their generation methods. We implemented Donald Knuth's Algorithm X in a efficient way in `C++` to construct balanced incomplete block designs. We were able to generate 1.104.000 (13-4-1) BIBDs under 4-5 seconds using this algorithm.

We have written this thesis based on several papers' claims about dropout designs that they could be more efficient than random method. This thesis acts as a resource for future research in this area. Since, the way we use balanced incomplete block designs in this thesis was just only a way over other many ways.

Applying dropout designs to the neural networks in a different way was introduced in a recent paper by *Chisaki, Shoko and Fuji-Hara, Ryoh and Miyamoto, Nobuko*. Although, this paper remained too theoretical and it didn't have implementations. The biggest disadvantage of this method was the unique and large hidden layer sizes it required for each different (v, k, λ)-BIBD.

2 COMBINATORIAL DESIGNS

Combinatorial design theory is a theory that has been used in many different computational fields related to the design and analysis of algorithms and hardware [2]. According to the widely accepted view, the modern studies on block designs began with the publication of an article by the statistician F. Yates in 1936. This article considered collections of subsets of a set with certain balance properties. One of the examples touched in the aforementioned article was :

Example 2.0.1. *Let* $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ *be 6-elements set and let* $\mathcal{A} = \big\{\{x_1, x_2, x_3\}, \{x_1, x_2, x_4\}, \{x_1, x_3, x_5\}, \{x_1, x_4, x_6\}, \{x_1, x_5, x_6\}, \{x_2, x_3, x_6\}, \{x_2, x_4, x_5\}, \{x_2, x_5, x_6\}, \{x_3, x_4, x_5\}, \{x_3, x_4, x_6\}\big\}$ *be the set of blocks. Here, we have 10 3-subsets of 6 element set* X *such that :*

1. *each block contains an equal number of elements which is 3 (=k) and*
2. *each distinct pair of elements appears exactly in 2 (=*λ*) different blocks.*

Second property is named balance of the design. Designs with this property called ***balanced incomplete block designs*** *(BIBD).*

Definition 2.0.1. *A BIBD is a collection of* k*-element subsets of a set* X *with* v *elements,* $k < v$*, such that each distinct pair of elements of* X *appears exactly in* λ *different blocks. We will denote this balanced incomplete block design by* $(X, \mathcal{A})$ *where* $\mathcal{A}$ *is the set of blocks.*

We can denote the number of blocks with the symbol b, and the block designs in short (v, k, λ). Therefore, the Example 2.0.1 is a (6,3,2)-BIBD.

Theorem 2.0.1. *In a* (v, k, λ)*-BIBD with* b *blocks each element occurs in* r *blocks where*

$$\lambda(v-1) = r(k-1) \tag{2.1}$$

and

$$kb = vr \tag{2.2}$$

Proof. Consider an element $x \in X$ where X is a v-set of (v, k, λ)-BIBD. x have $(v-1)$ other pairs in X and x occurs λ times as pair. Also x is repeated r times and it has pairs with $(k-1)$ other elements in a block. Therefore, $\lambda(v-1) = r(k-1)$. Secondly, each v element occurs r times and each block has k elements, hence $vr = bk$. □

Remark 2.0.1. *No (11,6,2)-design can exist.*

Proof. Theorem 2.0.1 is required for existance. Therefore a $(11, 6, 2)$-design must satify property 2.1 and 2.1 to exist but $b = 44/6$ is not an integer. □

Example 2.0.2. *This example will be called as* ***Fano Plane.*** *The set of blocks* $\mathcal{A} =$ *{{1,2,4}, {2,3,5}, {3,4,6}, {4,5,7}, {5,6,1}, {6,7,2}, {7,1,3}} forms a* $(7, 3, 1)$*-BIBD which can be considered as composition of the seven lines. A block design with* $k = 3$ *and* $\lambda = 1$ *is a Steiner triple system.*

In 1853, six years after a lot of research on Kirkman's paper about this subject, Steiner raised the question of the existence of such designs in a geometric context. For this reason, these $(v, 3, 1)$ systems are called Steiner triple systems.

Definition 2.0.2. *A Steiner triple system STS(v) of order* v *is a* $(v, 3, 1)$ *design.*

Lemma 2.0.2. *An STS(v) can exist only if* $v \equiv 1$ *or 3 (mod 6) [3].*

Definition 2.0.3. *A projective plane of order* n *is an* $(n^2 + n + 1, n + 1, 1)$ *design for* $n \geq 2$.

Therefore, the Fano Plane example is a projective plane of order 2 and a STS(7).

2.1 Incidence Matrices

In this section, the block designs will be represented by incidence matrices. The use of related matrices has greatly helped the study of block designs.

Definition 2.1.1. *The incidence matrix of a BIBD is a* $b \times v$ *matrix A, where b and* v *are the number of blocks and points respectively, such that* $A_{ij} = 1$ *if the point* j *is contained in ith block and 0 otherwise.*

Example 2.1.1. *For example, the incidence matrix of example of Yates (Example 2.0.1), can be expressed by the following incidence matrix :*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.3}$$

We can observe that rows correspond to the blocks and columns correspond to the elements. In addition, the incidence matrix may vary depending on the order of the blocks, but important features of the design will be independent of this. Example 2.1.1 is a $(6, 10, 5, 3, 2) - BIBD$.

Theorem 2.1.1. *Let N be the incidence matrix of a (v, k, λ) balanced incomplete block design, then $N^T N = (r - \lambda)I + \lambda J$ where I is a $v \times v$ identity matrix and J is a $v \times v$ matrix such that every entry is 1.*

Proof. By definition, N is a $b \times v$ matrix. Then $N^T N$ will be a $v \times v$ matrix. The entry (i, j) of matrix $B = N^T N$ is going to be the scalar product of the ith and jth columns of N. If $i = j$, b_{ij} is going to be the amount of 1s in the ith column, that is, the amount of blocks containing the ith element; therefore $b_{ii} = r$. If $i \neq j$, the scalar product will be the sum of 1s corresponding to rows in where columns i,j both have 1; as a result, b_{ij} is giving quantity of blocks that contains i'th and j'th elements together. That is, $b_{ij} = \lambda$. □

Theorem 2.1.2. *Let $(X, \mathcal{A})$ be a (v, k, λ) design, then the number of blocks in $\mathcal{A}$ is greater than or equal to the number of elements in X. In other words, $b \geq v$.*

Proof. Suppose N is the incidence matrix of given (v, k, λ) design. We will begin with showing that $N^T N$ is non-singular. That is, its determinant is non-zero. We will obtain

$$|N^T N| = \begin{vmatrix} r & \lambda & \lambda & \cdots & \lambda \\ \lambda & r & \lambda & \cdots & \lambda \\ \lambda & \lambda & r & \cdots & \lambda \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \lambda & \lambda & \lambda & \cdots & r \end{vmatrix} \tag{2.4}$$

$$= \begin{vmatrix} r & \lambda & \lambda & \cdots & \lambda \\ \lambda - r & r - \lambda & 0 & \cdots & 0 \\ \lambda - r & 0 & r - \lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \lambda - r & 0 & 0 & \cdots & r - \lambda \end{vmatrix}$$

if we substract the first row from each of the other rows. Now by adding the sum of all other columns to the first column, we will obtain

$$|N^T N| = \begin{vmatrix} r + (v-1)\lambda & \lambda & \lambda & \cdots & \lambda \\ 0 & r - \lambda & 0 & \cdots & 0 \\ 0 & 0 & r - \lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & r - \lambda \end{vmatrix} \tag{2.5}$$

$$= \{r + (v-1)\lambda\}(r - \lambda)^{v-1} = rk(r - \lambda)^{v-1}$$

on using theorem (2.0.1). But $k < v$ so by theorem (2.0.1), $r > \lambda$, therefore $|N^T N| \neq 0$. But $N^T N$ is a $v \times v$ matrix, so the rank $\rho(N^T N) = v$. Finally, since $\rho(N^T N) \leq \rho(N)$, and since $\rho(N) \leq b$, $v \leq \rho(N) \leq b$. □

Example 2.1.2. *There is not any (16,6,1) design existing since* $r = \frac{\lambda(v-1)}{k-1} = 3$*, so* $b = \frac{vr}{k} = 8 < v$*. Theorem (2.0.1) and Fisher's inequality (Theorem 2.1.2) are required for existance but they are not sufficient. (43,7,1) - design can be an example for this since it satisfies all conditions of Theorem 2.0.1 and Theorem 2.1.2 but there's no such design [4].*

Definition 2.1.2. *In a* (v, k, λ)*-BIBD, if* $b = v$*, the design is said to be symmetric.*

This is known as the extreme case of the Fisher inequality. The term *symmetry* does not come from the visual structure of the incidence matrix of the design, but from the

establishment of symmetry between the blocks and elements, as exemplified in the next theorem. The next theorem was proved for the first time in 1939 in Bose's paper.

Theorem 2.1.3. *If N is the incidence matrix of a symmetric design, then $NN^T = N^TN$ and every distinct pair of blocks intersect in λ elements.*

Proof. Note that $NJ = JN = kJ$, so that $N^TJ = (JN)^T = (kJ)^T = kJ$ for J, matrix of size $v \times v$ such that all entry is 1, and similarly, $JN^T = kJ$. Also, $J^2 = vJ$. Use will be made of the fact that a matrix commutes with its inverse. By equation (2.4),

$$\begin{aligned}(N^T - \sqrt{\frac{\lambda}{v}}J)(N + \sqrt{\frac{\lambda}{v}}J) &= N^TN + \sqrt{\frac{\lambda}{v}}(N^TJ - JN) - \frac{\lambda}{v}J^2 \\ &= N^TN - \lambda J = (k-\lambda)I.\end{aligned} \tag{2.6}$$

Thus $(1/(k-\lambda))(N + \sqrt{\frac{\lambda}{v}}J)$ is the inverse of $N^T - \sqrt{\frac{\lambda}{v}}J$ and hence commutes with it. Then,

$$\begin{aligned}(k-\lambda)I &= (N + \sqrt{\frac{\lambda}{v}}J)(N^T - \sqrt{\frac{\lambda}{v}}J) \\ &= NN^T + \sqrt{\frac{\lambda}{v}}(JN^T - NJ) - \sqrt{\frac{\lambda}{v}}J^2 \\ &= NN^T - \lambda J.\end{aligned} \tag{2.7}$$

Hence $NN^T = (k-\lambda)I + \lambda J = N^TN$ [5]. □

One way to get a new, different design using old designs is to replace each of the design's blocks with its complement.

2.2 Constructing New Designs

Definition 2.2.1. *Let D be a $(v, b, r, k, \lambda) - BIBD$ on a set X with v elements. Then the complementary design $\bar{D}$ will have its blocks as complements $X - B$ of the blocks B of D.*

Example 2.2.1. *Consider Fano Plane example (Example 2.0.2), then*

$$\bar{D} = \{\{3,5,6,7\},\{1,4,6,7\},\{1,2,5,7\},\{1,2,3,6\},\{2,3,4,7\},\{1,3,4,5\},\{2,4,5,6\}\} \tag{2.8}$$

is a (7,4,2) balanced incomplete block design.

Theorem 2.2.1. *Suppose that D is a (v, b, r, k, λ)-BIBD, then, $\bar{D}$ is a $(v, b, b-r, v-k, b-2r+\lambda)$ design provided that $b - 2r + \lambda > 0$.*

Proof. It is trivial that there are b blocks of size $v - k$ in $\bar{D}$. An element of X occurs in a block of $\bar{D}$ for sure when it does not occur in the corresponding block of D; so it occurs in $b - r$ blocks of $\bar{D}$. Finally, the number of blocks of $\bar{D}$ containing x and y is equal to the number of blocks of D that don't contain x or y. This number can be calculated as follows :

$= b -$ (amount of blocks in D that contains x or y)

$= b -$ (amount of blocks in D that contains x + contains y - contains both)

$= b - (r + r - \lambda) = b - 2r + \lambda.$

□

Corollary 2.2.2. *$\bar{D}$ is symmetric $(v, v-k, v-2k+\lambda)$ design if D is a symmetric (v, k, λ) design with $v - 2k + \lambda > 0$.*

Another way of obtaining new designs from old is throwing away all elements of one block B from all blocks $\mathcal{B} \in \mathcal{A}$ where $\mathcal{A}$ is the set of blocks. The obtained design will be called as **residual design**. The remaining blocks are all of size $k - \lambda$ and by theorem 1.1.3, any block of $\mathcal{A}$ intersects B in λ elements, therefore we obtain a new $(v - k, v - 1, r, k - \lambda, \lambda)$ design.

Definition 2.2.2. *Consider that $(X, \mathcal{A})$ is a symmetric (v, k, λ)-BIBD and let $B \in \mathcal{A}$. Then*

$$Res(X, \mathcal{A}, B) = (X \backslash B, \{A \backslash B : A \in \mathcal{A}, A \neq B\})$$

is called residual design obtained by deleting all points in a given block B.

We can obtain new design by deleting all the points not in a given block B and then deleting B aswell. The obtained new $(k, v - 1, k - 1, \lambda, \lambda - 1)$ design will be called as **derived design.**

Definition 2.2.3. *Consider that $(X, \mathcal{A})$ is a symmetric (v, k, λ)-BIBD, and let $B \in \mathcal{A}$.*

Then

$$Der(X, \mathcal{A}, B) = (B, \{A \bigcap B : A \in \mathcal{A}, A \neq B\})$$

is called derived design obtained by deleting all the points not in a given block B and then deleting B.

Theorem 2.2.3. *Let $(X, \mathcal{A})$ be a (v, k, λ) symmetric BIBD. Let B be one of the blocks of $\mathcal{A}$. If the $\lambda \geq 2$ condition is met, Der(X,A) becomes a $(k, v-1, k-1, \lambda, \lambda-1)$ BIBD. In addition, Res(X, A) becomes a $(v-k, v-1, k, k-\lambda, \lambda)$ BIBD if the condition $k \geq \lambda+2$ is met.*

Proof. $Der(X, \mathcal{A}, B)$ is a BIBD as long as $k > \lambda \geq 2$ (k is the number of points in the derived design, and the blocks have size λ). Nonetheless, $k > \lambda$ in any symmetric BIBD since $\lambda(v-1) = k(k-1)$ is provided by Theorem 2.0.1 and $v > k$. So, this condition is trivial.

$Res(X, \mathcal{A}, B)$ is a BIBD as long as $v - k > k - \lambda \geq 2$ ($(v-k)$ is the number of points in a residual design and the blocks have size $k - \lambda$). Starting to proof by showing that $v - k > k - \lambda$ is true in a symmetric BIBD. Consider that $v \leq 2k - \lambda$; then we have $k(k-1) = \lambda(v-1) \leq \lambda(2k - \lambda - 1)$. This is identical to $(k-\lambda)(k-\lambda-1) \leq 0$. But k and λ are integers, so this last inequality holds if and only if $k = \lambda$ or $k = \lambda + 1$. We assumed that $k \geq \lambda + 2$, so there is a contradiction. Accordingly, the condition $v - k > k - \lambda$ is satisfied. □

Example 2.2.2. $(11, 5, 2)$ *is a symmetric BIBD owing to the fact that $b = v = 20$. A residual $BIBD$ obtained from this symmetric design will be a $(6, 3, 2) - BIBD$ and a derived BIBD will be (5,2,1)-BIBD.*

There are plenty of ways of obtaining new balanced incomplete blocks. Paper of *S.S. Shrikhande* sheds light on this, introduces new

2.3 Isomorphisms of Designs

In this section, we will explain whenever two balanced incomplete block designs are considered as isomorph block designs.

Definition 2.3.1. *If there exists a one one-to-one mapping from the set of elements of D_1 to the set of elements of D_2, D_1 and D_2 are same or isomorphic where D_1 and D_2 are two (v, b, r, k, λ) designs. Otherwise D_1 and D_2 are different designs.*

Example 2.3.1. *The following $(7, 3, 1)$ design :*

$$D_1 = \{\{1, 2, 3\}, \{1, 4, 5\}, \{1, 6, 7\}, \{2, 4, 6\}, \{2, 5, 7\}, \{3, 4, 7\}, \{3, 5, 6\}\}$$

is isomorphic to the following (7-3-1) design D_1 for the mapping $1 \to 2, 2 \to 1, 3 \to 4, 4 \to 7, 5 \to 6, 6 \to 3, 7 \to 5$.

$$D_2 = \{\{2, 1, 4\}, \{2, 6, 7\}, \{2, 3, 5\}, \{1, 7, 3\}, \{1, 6, 5\}, \{4, 7, 5\}, \{4, 6, 3\}\}.$$

We can also see this from incidence matrices aswell with changing row 1 with row 2, row 3 with 4, 5 with 6, 6 with 3, 4 with 7, 7 with 5 of N_1. What we obtain will be N_2, incidence matrix of D_2. Therefore $N_1 \cong N_2$.

3 CONSTRUCTION OF BIBDS USING ALGORITHM X

In this chapter, starting from the exact cover problem, to Algorithm X, we will construct our balanced incomplete block designs. First, we explain the exact cover problem, then, we move into Algorithm X that acts like a tool at solving exact cover problems. Afterwards, we explain our algorithm that uses Algorithm X to generate BIBDs.

3.1 Exact Cover Problem

Let B be the collection of subsets of set X then $B' \in B$ is an exact cover of X if x contained in exactly one subset of B' for all $x \in X$ and $S \cap S' = \emptyset$ for all $S, S' \in B'$ where $S \neq S'$ and union of subsets S of B' is X.

Example 3.1.1. *Let* $B = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$ *and* $X = \{1, 2, 3, 4, 5, 6, 7\}$ *such that*

- $\mathcal{A} = \{1, 2, 3, 4\}$
- $\mathcal{B} = \{1, 2, 3, 6\}$
- $\mathcal{C} = \{4, 5, 7\}$
- $\mathcal{D} = \{5, 6, 7\}$

Then $B' = \{\mathcal{A}, \mathcal{D}\}$ *and* $B'' = \{\mathcal{B}, \mathcal{C}\}$ *are exact covers.*

Now lets look into the following incidence matrice, finding exact covers of this matrice would be answering the question, does this matrix have a set of rows containing exactly one 1 in each column ?

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.1}$$

This matrix has such a set (rows 2,4,7). This is a well-known hard problem that is NP-complete even if each row contains a precisely certain amount of 1s.

3.2 Algorithm X

Before explaining Donald Knuth's dancing links, we shall give definition of a doubly linked list and a brief reasoning about why did we choose Algorithm X.

Why Algorithm X ?

Algorithm X with dancing links, called Algorithm DLX is working as fast as other backtracking algorithms in *small* cases but when it comes to the *large* cases, Algorithm X appears to run faster than other special-purpose backtracking algorithms (such as Dijkstra's or Hitotumatu and Noshita's backtracking algorithms) because of its ordering heuristic [7]. In our thesis, we generated $(v, k, 1)$ designs using Algorithm X and for some BIBDs (like (15,3,1) or (13,4,1)), it was a large case. Therefore, we decided using Algorithm X to have quicker solutions.

Also, the idea of (3.3) (putting x back to the doubly linked list with uncovering process) was first introduced in 1979 by Hitotumatu and Noshita [8], who showed that it makes Dijkstra's well-known program for the N queens problem [9, pages 72–82] run nearly twice as fast without making the program significantly more complicated. For further efficiency considerations of Algorithm X and speed comparisons of algorithms, you may see Donald Knuth's dancing links paper.

Definition 3.2.1. *In computer science, a doubly linked list is a linked data structure consisting of a sequentially linked set of records called nodes. Each node contains three fields : two link fields (references to previous and next nodes in the node array) and a data field. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a node or null, to make traversal in the list easier. And if there is only one node in the doubly linked list, then the list is circularly linked with its only node [10].*

These previous and next node fields given in definition (3.2.1) will be represented with $L[x]$ and $R[x]$ respectively where x is representing a node in doubly linked list in the next section.

3.2.1 Dancing Links

In this section, we will introduce Donald Knuth's dancing links which are going to be used later in Algorithm X [7].

Suppose x points to an element (*node*) of a doubly linked list; let $L[x]$ and $R[x]$ point to the predecessor and successor of that element. Then the operations :

$$L[R[x]] \leftarrow L[x],\ R[L[x]] \leftarrow R[x] \tag{3.2}$$

remove x from the list. Also subsequent operations :

$$L[R[x]] \leftarrow x,\ R[L[x]] \leftarrow x \tag{3.3}$$

will place x back into the list again. This process with pointers was called as "dancing links" by Donald Knuth.

3.2.2 Solving an exact cover problem using Algorithm x

Algorithm X is simply a trial-and-error approach. Pseudo code of the algorithm for a given matrix A of 0s and 1s is as follows :

- If matrix A is empty, the problem is solved, we can terminate successfully. Otherwise we choose a column, c (deterministically).
- Choose a row, r, such that $A[r, c] = 1$ (nondeterministically).
- Include r in the partial solution.
- For each j such that $A[r, j] = 1$,
  - delete column j from matrix A.
  - for each i such that $A[i, j] = 1$,
    - delete row i from matrix A.
- Repeat this recursively on the reduced matrix A.

3.2.3 The dance steps

To be able to apply Algorithm X, we will represent each 1 in matrix A as data objects with five components. These components will be represented by the symbols $L[x], R[x], U[x], D[x], C[x]$. The rows of matrix A are circular lists that are connected by two sides and with L and R components. On the other hand, the columns of the matrix are lists that are connected at both ends and have U and D components.

List headers are part of a larger object called a column object. These column objects have R, L, D, U, C components as well as N, S components. The S component points to the number of 1s in the column object and we can use this to print the results. The C component of each data object in A points to the mentioned column object.

The R and L components in the list headers connect all columns that are not yet covered and should be covered. The circular lists resulting from this linking also contain the custom column object called root, h, which acts as the main header for all other active headers. $D[h], C[h], U[h], S[h]$ and $N[h]$ components are not used.

Donald Knuth's non-deterministic algorithm for finding all exact covers, Algorithm X, can now be initialized with the following $search(k)$ function, which is a recursive procedure initialized with a value of k=0. The pseudo code of $search$ function is as follows :

- If header's right equals to header component, print the current solution and return.
- Else, select a column object c.
- Cover the selected column c.
- Go down starting from selected column until returning back to the selected column using the following : $r \leftarrow D[c], D[D[c]], ...,$ while $r \neq c$, and for each visited row, do the following :
  - set $O_k \leftarrow r$;
  - for each $j \leftarrow R[r], R[R[r]], ...,$ while $j \neq r$,
    - cover column j

- $search(k+1)$;
- set $r \leftarrow O_k$ and $c \leftarrow C[r]$;
- for each $j \leftarrow L[r], L[L[r]], ...,$ while $j \neq r$,
  - uncover column j ;

— Uncover column c and return.

To print the current solution, we print the rows containing $O_0, O_1, ..., O_{k-1}$, where the row containing data object O is printed by printing $N[C[O]], N[C[R[O]]], N[C[R[R[O]]]]$....

To select a column object c, we can set $c \leftarrow R[h]$, this will be giving left most uncovered column. Or, as a second method, we can set $s \leftarrow \inf$ and

- for each $j \leftarrow R[h], R[R[h]], ...,$ while $j \neq h$,
  - if $S[j] < s$, set $c \leftarrow j$ and $s \leftarrow S[j]$.

After this process, c will be a column with smallest number of 1s. To cover a column c,

- Set $L[R[c]] \leftarrow L[c]$ and $R[L[c]] \leftarrow R[c]$,
- For each $i \leftarrow D[c], D[D[c]], ...$ while $i \neq c$,
  - for each $j \leftarrow R[i], R[R[i]], ...,$ while $j \neq i$,
    - set $U[D[j]] \leftarrow U[j]$, $D[U[j]] \leftarrow D[j]$,
    - and set $S[C[j]] \leftarrow S[C[j]] - 1$.

Operations used here to remove objects in both the horizontal and vertical directions.

To uncover a column c,

- For each $i = U[c], U[U[c]], ...,$ while $i \neq c$,
  - for each $j \leftarrow L[i], L[L[i]], ...,$ while $j \neq i$,
    - set $S[C[j]] \leftarrow S[[j]] + 1$,
    - and set $U[D[j]] \leftarrow j, D[U[j]] \leftarrow j$.
- Set $L[R[c]] \leftarrow c$ and $R[L[c]] \leftarrow c$.

Notice that uncovering takes place in precisely reverse order of the covering operation. These dancing link operations explained more detailed in Donald Knuth's paper [7].

3.3 Finding BIBDs using Algorithm X

In this chapter, we will use Algorithm X to find all possible (v, k, λ) BIBDs. To do this, we will follow the steps :

— Building incidence matrix for given (v, k, λ) to apply Algorithm X on.

— Apply Algorithm X to incidence matrix generated previously.

From now on, we take $(v, k, 1)$ BIBDs as examples, that is, $\lambda = 1$ cases.

3.4 Generation of the incidence matrix

We want to build the incidence matrix we need, naming I, using the following steps :

— Generate all k-element combinations of $\binom{v}{k}$, these will be our rows.

— Generate all 2-element combinations of $\binom{v}{2}$, these will be our columns.

— Fill the incidence matrix.

We will fill the incidence matrix I based on the following rule :

$$I[i,j] = \begin{cases} 1 & \text{if column j is a subset of row i} \\ 0 & \text{else} \end{cases} \qquad (3.4)$$

where $1 \leq i \leq \binom{v}{k}$ and $1 \leq j \leq \binom{v}{2}$.

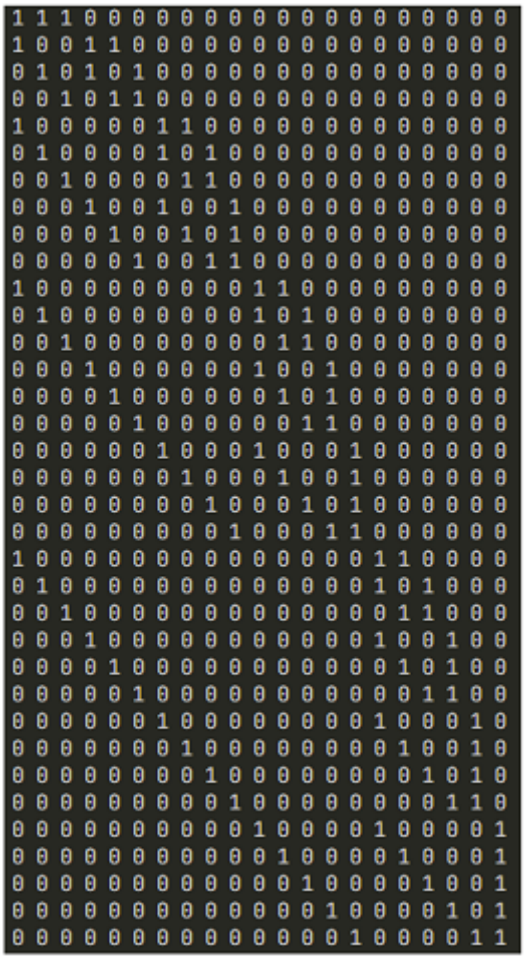


FIGURE 3.1 – Incidence matrix generated for parameters $v = 7, k = 3$

Example 3.4.1. *Consider Fano Plane example, that is,* $v = 7, k = 3, \lambda = 1$. *For Fano Plane, all 3 element combinations* $\binom{7}{3}$ *of incidence matrix I will be :*

$\Big\{$ *{5, 6, 7},{4, 6, 7},{4, 5, 7},{4, 5, 6},{3, 6, 7},{3, 5, 7},{3, 5, 6}, {3, 4, 7},{3, 4, 6},{3, 4, 5},{2, 6, 7},{2, 5, 7},{2, 5, 6},{2, 4, 7},{2, 4, 6}, {2, 4, 5},{2, 3, 7},{2, 3, 6},{2, 3, 5},{2, 3, 4},{1, 6, 7},{1, 5, 7},{1, 5, 6},{1, 4, 7},{1, 4, 6},{1, 4, 5},{1, 3, 7},{1, 3, 6},{1, 3, 5},{1, 3, 4},{1, 2, 7},{1, 2, 6},{1, 2, 5},{1, 2, 4},{1, 2, 3}* $\Big\}$.

And the all 2 element combinations $\binom{7}{2}$ *of I will be :*

$\Big\{$ *{6, 7},{5, 7},{5, 6},{4, 7},{4, 6},{4, 5},{3, 7},{3, 6},{3, 5},{3, 4},{2, 7},{2, 6},{2, 5},{2, 4},{2, 3},{1, 7},{1, 6},{1, 5},{1, 4},{1, 3},{1, 2}* $\Big\}$.

Therefore for example $I[1,1] = 1$ *since* $\{6,7\}$ *is a subset of* $\{5,6,7\}$ *and so on. After filling* I *completely, we will obtain incidence matrix shown in Figure 2.1*

3.5 Applying Algorithm X to the generated incidence matrix

In generated matrix I, each row contains exactly v 1s and the set of rows containing exactly one 1 in each column will have $b = \frac{(\lambda(v-1)/(k-1))v}{k}$ number of rows in it, which is 7 for Fano Plane example. We have proven this formula in Chapter 2.

After obtaining the incidence matrix for given v, k, λ, we simply apply Algorithm X to it, starting with $search(0)$. Each solution will contain b number of column ids.

Example 3.5.1. *For Fano Plane example, if we apply Algorithm X to the incidence matrix shown in Figure 2.1, we will have results shown in Figure 2.2.*

```
alicsharp@DESKTOP-AG99MAC:/mnt/d/TEZ/kodlar/BIBDs_via_AlgorithmX/src$ ./a.out
1 8 31 15 26 28 19
1 8 31 25 16 18 29
1 14 27 9 26 19 32
1 14 27 25 10 18 33
1 24 17 9 16 29 32
1 24 17 15 10 28 33
2 6 31 13 26 28 20
2 6 31 23 16 18 30
2 12 27 7 26 20 32
2 12 27 23 10 18 34
2 22 17 7 16 30 32
2 22 17 13 10 28 34
5 3 31 13 25 29 20
5 3 31 23 15 19 30
5 12 24 4 29 20 32
5 12 24 23 15 10 35
5 22 14 4 19 30 32
5 22 14 13 25 10 35
11 3 27 7 25 20 33
11 3 27 23 9 19 34
11 6 24 4 28 20 33
11 6 24 23 9 16 35
11 22 8 4 28 19 34
11 22 8 7 25 16 35
21 3 17 7 15 30 33
21 3 17 13 9 29 34
21 6 14 4 18 30 33
21 6 14 13 9 26 35
21 12 8 4 18 29 34
21 12 8 7 15 26 35
30 solutions found for (7,3,1) design.
```

FIGURE 3.2 – (7,3,1) BIBDs

In Figure 2.2, each line in the output represent a solution, that is,
1 8 31 15 26 28 19 is first solution found. Also the numbers 1, 8, 31, 15, ... represent row ids. That is for first solution, the set containing row 1, row 8, ...,row 28 and row 19 contains exactly one 1 in each column. For first solution and for incidence matrix generated as figure 2.1, the set containing rows {1,8,31,15,26,28,19} represent the following BIBD :

$$\{\{5,6,7\},\{3,4,7\},\{1,2,7\},\{2,4,6\},\{1,4,5\},\{1,3,6\},\{2,3,5\}\}$$

where row 1 represent {5,6,7}, row 2 represent {3,4,7}, ..., row 19 represent {2,3,5} respectively. Finding set of rows containing exactly one 1 in each column is equivalent to solving linear equation $IX = 1$ where $\lambda = 1$. The operations done above was for case λ=1. Then, for cases λ>1, we shall solve the linear equation $IX = \lambda$ instead of $IX = 1$.

4 DEEP NEURAL NETWORKS

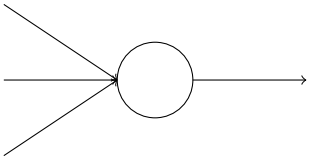
In this chapter we will explain deep neural networks, starting from neural networks.

What are neural networks ?

Neural networks, also known as artificial neural networks are a subset of machine learning. ANNs form the basis of deep learning algorithms. We are using similar figures or terminologies while talking about human brains. Because, artificial neural networks' structure and name are inspired by the human brain.

4.1 Perceptrons

To get started with neural networks, we can start with the most basic neuron called a perceptron. The perceptron is the oldest neural network, created by Frank Rosenblatt in 1958, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it is more common to use other models of artificial neurons. Rosenblatt proposed a simple rule to compute the output. He introduced weights, $w1, w2, \ldots$, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted $\sum_j w_j x_j$ is less than or greater than some threshold value [1]. A perceptron takes several binary inputs, $x1, x2, \ldots$, and produces a single binary output :



To explain in algebraic terms, $output = 0 if \sum_j w_j x_j \leq$ the threshold and 1 otherwise. Threshold value may vary depending on the purpose.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold.} \end{cases}$$

Neural networks gives outputs similar to this are consisting of several such perceptrons. These neural networks are called *feedforward neural networks*, or multi-layer perceptrons

(MLPs). They are comprised of an input layer, a hidden layer or layers, and an output layer [1].

4.2 Improving the way neural networks learn

There are several methods to make neural network learn better. In this thesis, we will mention the regularization technique called dropout method.

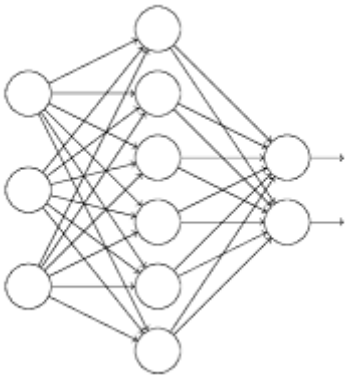
4.2.1 Overfitting

The main purpose of learning is to gain the ability to make an accurate prediction on test data, not on training data. Models that can make good predictions on training data but not on test data are called overlearned models and this situation is called *"overfitting"*. Overfitting occurs when the model learns the details and noises in the training data. Since an overlearning model can easily predict the training data, the models are tested with test data consisting of different data from the training data. Errors encountered in test data are called test errors and it is expected that as the training errors decrease, the test errors should also decrease. This overfitting can be prevented by randomly omitting some percentage of the feature detectors (neurons in the hidden layers) on each training case. Also the "dropout" method gives big improvements on many benchmark tasks and sets new records for speech and object recognition [11].

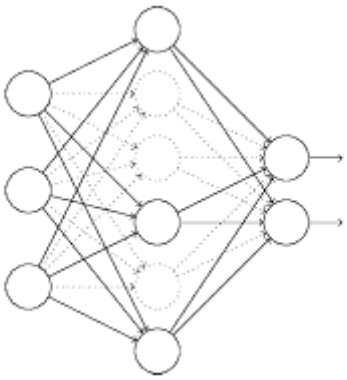
4.2.2 Regularization Technique : Dropout

Dropout is a completely different technique from other regularization techniques. When we look at L1 and L2 regularization techniques, we see that they are trying to change the cost function [12]. However, unlike L1 and L2, the dropout technique changes the neural network itself. Now, we will first explain the basics of the dropout technique and then answer how it works [1].

Suppose we're trying to train following network :



Suppose x are training inputs and y are expected outputs corresponding to these inputs. In the absence of the dropout method, we would train this neural network with forward and backward propagations without changing it, but this process is slightly different when the dropout method is present. We start the training by randomly and temporarily deleting a certain percentage of the hidden neurons in the network without touching the neurons in the input and output layers. After doing this, we will get a mesh similar to the one below. Note that the dropped neurons, that is the temporarily deleted neurons, are still in the neural network as ghosts :



We propagate the input x forward through the neural network where we randomly left some neurons earlier, and then we propagate the result back through the same neural network. Then we update the appropriate weights and biases. Next, for next entry to the neural network, we repeat the same process by first restoring the ghosted neurons, then choosing a new random subset of hidden neurons to delete and so on.

Repeating this process multiple times will teach our network a set of weights and biases. When we run the full network, more neurons will become active as the percentage of the neurons we ghosted, so to compensate for this, we increase the weights of the neurons

in the trained neural network by dividing the weights by the percentage of ghosts in the neural network.

4.2.3 Random Dropout

In this section, we will try to clarify the random method in weight estimation. After explaining random method, we will represent ghosted neural network by corresponding incidence matrice. The process of selecting weights in neural network randomly is called *random dropout*. Now let's consider a small model with 5 nodes in the input layer and 3 nodes in the hidden layer as an example. Then, after taking our inputs (x_1, x_2, x_3, x_4) and x_5, we can determine our outputs u_1, u_2 and u_3 with the following calculation :

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \tag{4.1}$$

Let's focus only on u_1 for now, to explain our topic more clearly. Then, the equality for u_1 will be as follows :

$$u_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 + w_{15}x_5 + \epsilon_1$$

where ϵ_1 is the margin of error. Now let's call $x_i^{(j)}$ variable x_i of the jth input data, $\epsilon_1^{(j)}$ the margin of error of the jth input data, and $u_1^{(j)}$ the output determined by jth input data [13].

Now assume that the weights w_{11}^j, $w_{12}^{(j)}$, ..., $w_{15}^{(j)}$ are randomly dropped out for each input. Then we calculate our outputs $u_1^{(1)}, u_1^{(2)}, ...$ for the input data as follows :

$$\begin{pmatrix} u_1^{(1)} \\ u_1^{(2)} \\ u_1^{(3)} \\ \vdots \end{pmatrix} = \begin{bmatrix} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} & \\ x_1^{(2)} & & & & x_5^{(2)} \\ & x_2^{(3)} & & x_4^{(3)} & \\ & & & \vdots & \end{bmatrix} \begin{pmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{14} \\ w_{15} \end{pmatrix} + \begin{pmatrix} \epsilon_1^{(1)} \\ \epsilon_1^{(2)} \\ \epsilon_1^{(3)} \\ \vdots \end{pmatrix} \tag{4.2}$$

The process above is a regression model with sparse data. That being so, it can be demonstrated by the following incidence matrix.

$$X = \begin{bmatrix} & 1 & 1 & 1 & \\ 1 & & & & 1 \\ & 1 & & 1 & \\ & & \vdots & & \end{bmatrix}$$

Remark 4.2.1. *Similar to the process done here, we will represent our neural networks by incidence matrices in the following chapters. Representing our artificial neural networks by incidence matrices will make it easier for us when we want to apply the block designs to our neural networks in the next sections owing to the fact that block designs can also be represented by corresponding incidence matrices. Incidence matrices are used in many fields, such as here, in the representation of block designs and neural networks.*

4.3 Deep Learning

The structure of artificial neural networks is suitable for many methods of deep learning. Neural networks in which deep learning models are used are called deep neural networks. The word deep comes from the fact that there are more hidden layers in between than in a normal neural network. The number of hidden layers in deep neural networks can even go up to 150. Thanks to the deep learning model, we can extract results directly from the data and the manual feature extraction used in normal machine learning may not be needed here.

Consider the following multi-layer neural network :

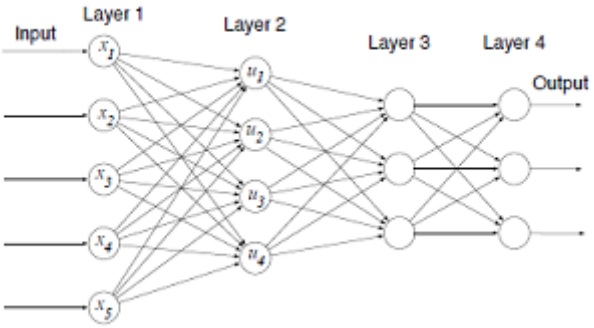


FIGURE 4.1 – A NN With Multiple Layers [1]

There are weights w_{ij} on connections between neurons, and these weights reveal the importance of the input value. Initially, these weights are determined randomly. Input values and output values are defined as $x = [x_1, x_2, ..., x_n]$, and $U = [u_1, u_2, ..., u_m]$ respectively. W is a matrix of $m \times n$ size, containing weights. The output values $U = [u_1, u_2, ..., u_m]$ are calculated based on the input data as follows :

$$U = \alpha(\mathbf{W}x + \mathbf{b}) \tag{4.3}$$

where b is the deviation value called *bias* and α is the activation function used in that layer stage. The different layer stages are calculated in the same way and activation functions may vary between stages of layers.

5 BUILDING DROPOUT MODEL

In this section, we will first create our sample machine learning model that uses dropout design regularization. Next, we'll explain how we train this machine learning model and how we predict new data. Also, while doing this, we will not forget to mention what our dataset looks like.

5.1 Sample Problem to Apply Model

The next example problem is a classic one that can also be found in many machine learning courses on the internet. The data used in our thesis was taken from a Coursera lecture which is given by Prof. Andrew Ng [14]. You can enlarge the dataset for this problem by generating more random values, or shrink it by subtracting some of its values.

Problem Statement : Suppose you are hired by the Football Corporation as an AI specialist and they ask you to suggest positions where your team keeper should throw the ball so that players on your team can then head off. To do this, they provide you with the following 2D training dataset saved from previous games :

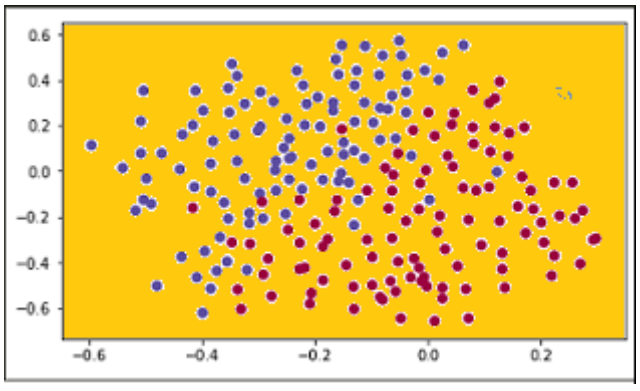


FIGURE 5.1 – Football field

The cells in Figure (5.1) correspond to the positions in which the players in the team hit the ball with their heads after the goalkeeper of your team has taken a shot.

— If the dot is blue, your team's player can hit the ball with his/her head.

— Otherwise, we assume that he/she cannot hit.

Our goal : Using a deep learning model with dropout designs regularization method to find the positions on the field where the goalkeeper should kick the ball.

To reach our goal, we will follow the steps given below :

— Achieving a good model by fine-tuning model parameters such as learning rate, number of iterations, number of layers, or by changing model settings such as changing the layer sizes used or the regularization technique used. Reaching a good model goes through trial and error.

— Training our model using our train dataset. In our case, we will apply dropout regularization technique at this stage to prevent overfitting.

— While applying dropout technique, we will use balanced incomplete block designs instead of using random method.

— After updating our parameters W and b in training process, we will predict the test data, which is seperated from training data.

— We will measure model's accuracy in training and test sets to measure model's success.

In machine learning, building proper model for your train and test datasets, and for your objective is essential. In our case, we will try applying (7,3,1)-BIBDs to our neural network. And to do this, we will construct a deep neural network with seven hidden layers. (See figure 5.2)

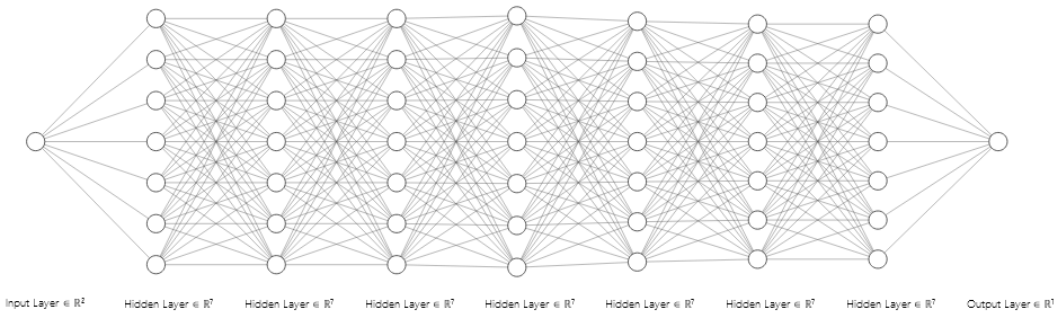


FIGURE 5.2 – Neural Network with Seven Hidden Layers

At the beginning, we are turning on all nodes. We will turn nodes off later depending on the given block design's structure. Consider that we will apply the following (7,3,1)-BIBD to the neural network shown in figure 5.2 where $(X, \mathcal{A})$, the set of elements and

$\mathcal{A} = \{\{1,2,5\},\{1,3,6\},\{1,4,7\},\{2,3,7\},\{2,4,6\},\{3,4,5\},\{5,6,7\}\}$, the set of blocks. This will make us to close 1st, 2nd and 5th nodes of first hidden layer, 1st, 3rd and 6th of 2nd hidden layer ... and 5th, 6th, 7th nodes of the 7th hidden layer. After turning those nodes off, we will have a neural network similar to the following :

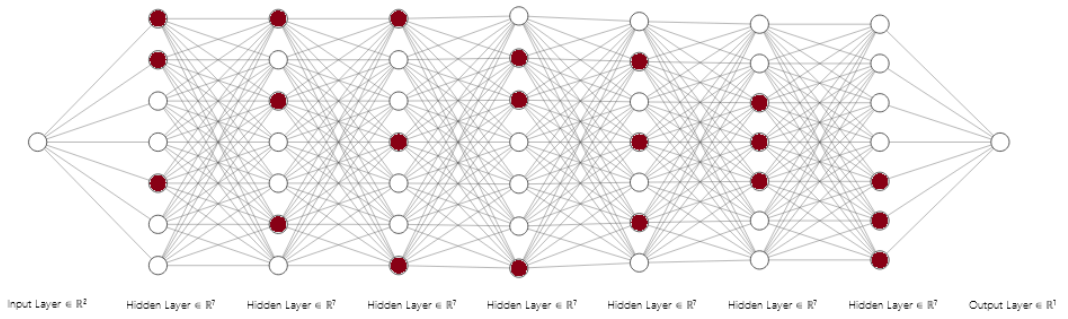


FIGURE 5.3 – Dropout Used Deep Neural Network

Turning off node in our neural network is done by giving them zero weight in that turn. That is, incoming and outgoing weights to the colored nodes in figure 5.3 will be zero.

5.2 Training Model and Predicting New Input

We have a training data X of size (2,211). That is, there are 211 inputs and each input is corresponding to a point (x, y) on the field. We are training our model for 30.000 iterations. At the beginning, before starting the iterations, we are initializing our core parameters W and b. Then, each iteration has following steps :

— Apply forward propagation and return parameters U, A where $U = Wx + b$ and $A = \alpha(U)$ where α is the activation function used in that layer.

— Compute cost of current iteration by the following formula :

$$cost = \frac{1}{m}(\sum(\log(A) * Y) + (\log(1 - A) * (1 - Y))) \qquad (5.1)$$

where operator $*$ is used for element-wise multiplication, Y is test dataset and m is Y's size, an integer.

— Run backward propagation.

— Update parameters W and b using the following formula :

$$W = W - \gamma(dW) \tag{5.2}$$

and

$$b = b - \gamma db \tag{5.3}$$

where γ is learning rate and dW, db are gradients we calculated during backward propagation.

The parameters W and b are being updated for number of iteration times. At the end of iterations, our model is becoming ready to predict new data with its updated W and b. In our network, RELU activation functions are used in hidden layers and sigmoid activation function is used for our output layer since our output layer's type is binary, i.e., for given test point (x, y), is it a good place for the goalkeeper to shoot or not (i.e. red or blue). Their formulas are as following :

$$sigmoid(z) = \frac{1}{e^{-z}} \tag{5.4}$$

$$RELU(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{5.5}$$

where $0 \leq sigmoid(z) \leq 1 \ \forall z \in \mathbb{C}$.

Remark 5.2.1. *Note that other activation functions could have been used in this neural network's hidden layers and output layer.*

6 COMPUTATIONAL STUDIES

In this chapter, two main computational studies, implementation of Algorithm X and implementation of machine learning models will be explained. While explaining the implementations, some computational details and codes will also be given.

6.1 Exact Cover Problem

Algorithm X was implemented to generate balanced incomplete block designs for given parameters (v, k, λ). This process was explained in chapter 3. Here computation speed of the algorithm for several parameters will be shared. Algorithm is implemented in `C++` programming language. First, it was tested in Python but in Python, the time it takes to solve exact cover problem was very long compared to what we could have in `C++` .

Remark 6.1.1. *The BIBDs generated through algorithm may be isomorph to each other. That is, they all are not distinct from each other. For example, all of 30 (7,3,1)-BIBDs generated are isomorph to each other [15]. It is possible to prevent generating isomorph BIBDs by integrating Nauty, graph isomorphism algorihm, into the algorithm we built [16]. By this, we could simply avoid isomorph BIBDs. Secondly, algorithm's current version at GitHub is working for cases where* $\lambda = 1$.

We will generate 30-(7,3,1) BIBDs under 17ms.

```
alicsharp@DESKTOP-AG99MAC:/mnt/c/Users/ali.ozturk/c++_projects/BIBDs_via_AlgorithmX/src$ time g++ main.cpp Combinations.cpp IncidenceMatrix.cpp

real    0m2.572s
user    0m1.609s
sys     0m0.922s
alicsharp@DESKTOP-AG99MAC:/mnt/c/Users/ali.ozturk/c++_projects/BIBDs_via_AlgorithmX/src$ time ./a.out
30 solutions found for (7,3,1) design.

real    0m0.015s
user    0m0.000s
sys     0m0.016s
alicsharp@DESKTOP-AG99MAC:/mnt/c/Users/ali.ozturk/c++_projects/BIBDs_via_AlgorithmX/src$
```

FIGURE 6.1 – (7,3,1) BIBD Algorithm Speed

These 30 (7,3,1) BIBDs are given in (Figure 3.2).

1.108.000 (13,4,1)-BIBD can be generated with Algorithm X under 6 seconds. These 1.108.000 solutions can be found on project's repository at GitHub.

FIGURE 6.2 – (13,4,1) BIBD Algorithm Speed

Further work can be done using this algorithm but for now, we will use the balanced incomplete block designs we generated in our deep learning models.

6.2 Saving Solutions, Generated BIBDs

In this section, we explain how and when we do write to the text file that stores BIBDs. After starting to the search process in Algorithm X, we are checking if header's right is equal to the header itself, because if this is the case, we found a solution there. So, each time this happens while in search function, we are writing the found BIBD to the text file.

6.2.1 Writing Process

The solution found will consist of b distinct column ids. In our case, while searching for $(7, 3, 1)$-designs, the solution will consist of $b = 7$ (we have proven this in Theorem 2.0.1) distinct column ids and these $b = 7$ column ids are integers between 1 and $\binom{7}{3} = 35$. We explained what column ids represent detailed in Chapter (3.5). At the start of search function, we are checking if header's right is header itself. If this is the case, we are writing the column ids of the solution to the text file. Figure (3.2) is an example of the column ids of the solutions.

6.3 Machine Learning

In this section, dropout design model's, random dropout model's and model without dropout's results will be shared and dropout design model's results will compared with other dropout methods' results. These computational work is done on sample dataset which is explained in (5.1). We are training our models with same sample data X of size (2,211). Learning rate(γ) were 0.05 and 0.03 for 30.000 iterations in all these simulations.

6.3.1 Model Without Dropout Regularization

In this subsection, we will be sharing our simulation results of model that has no regularization technique applied. That is, our 7-hidden-layer deep neural network's nodes will be turned on always on each iterations. First, lets share its accuracies on train and test datasets.

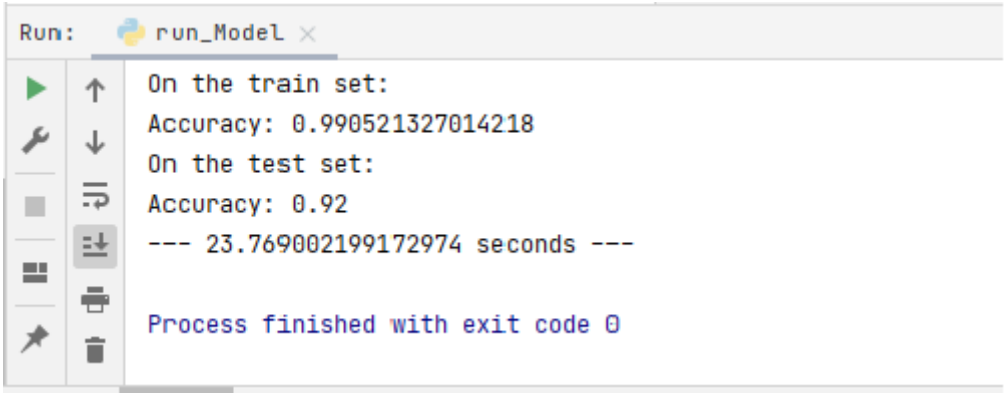


FIGURE 6.3 – Model Without Dropout Regularization

We have 99% accuracy on train dataset where we have only 92% accuracy on test dataset. This is called overfitting. This can be seen from the colored graph of train data (Figure 6.4) aswell.

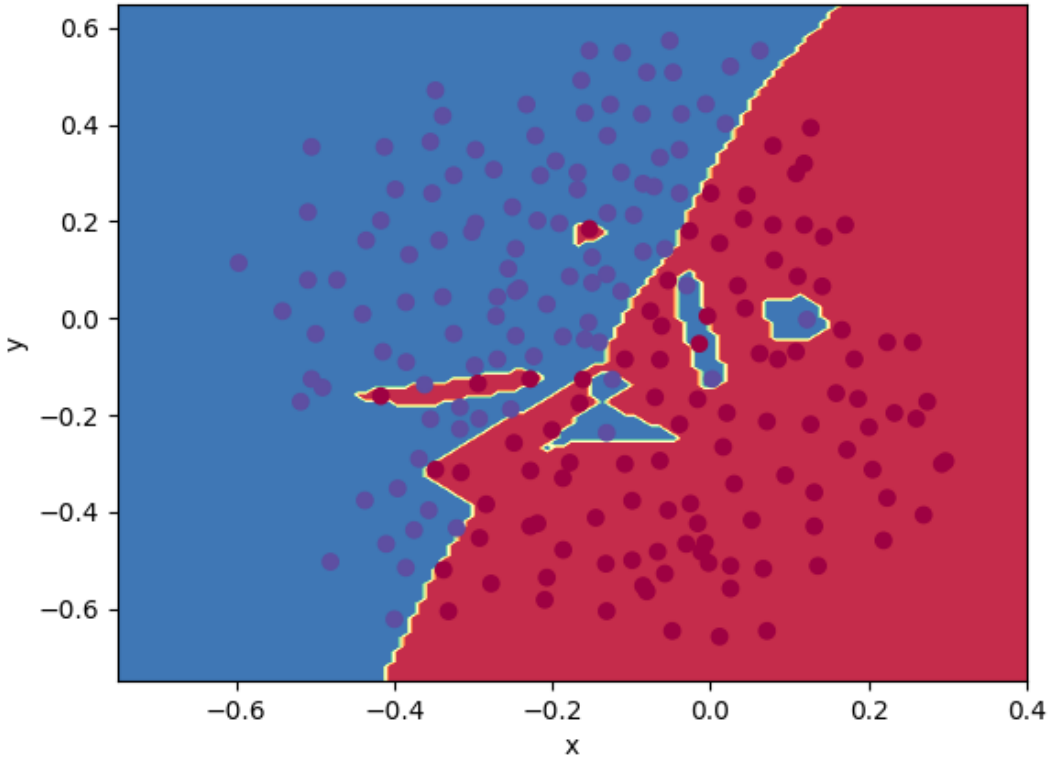


FIGURE 6.4 – Model Without Dropout Regularization - Figure 6.3's Graph

We want to avoid having this result. It is always better to have better accuracy on test dataset. To prevent this, as it is mentioned earlier, we will drop nodes in our 7-hidden layered deep neural network as shown in Figure (5.3).

6.3.2 Model With Dropout Designs Regularization

In this subsection, we will be sharing our simulation results of model that has dropout designs as regularization technique. That is, our 7-hidden-layer deep neural network's nodes will be turned off on each iteration based on given balanced incomplete block design's incidence matrix. First, lets share its accuracies on train and test datasets.

```
Run:   run_Model
Accuracy: 0.909952606635071
On the test set:
Accuracy: 0.94
--- 33.01512932777405 seconds ---
Design used in this run: [[1, 2, 5], [1, 3, 6], [1, 4, 7], [2, 3, 7], [2, 4, 6], [3, 4, 5], [5, 6, 7]]

Process finished with exit code 0
```

FIGURE 6.5 – Model With Dropout Designs Regularization

We have 90% accuracy on train dataset and we have 94% accuracy on test dataset. We prevented overfitting in this simulation using the following (7,3,1)-BIBD or one of its permutations on each iteration for 30.000 iterations.

$$\{\{1,2,5\},\{1,3,6\},\{1,4,7\},\{2,3,7\},\{2,4,6\},\{3,4,5\},\{5,6,7\}\}$$

The colored graph we are having in training dataset will be as following in this case :

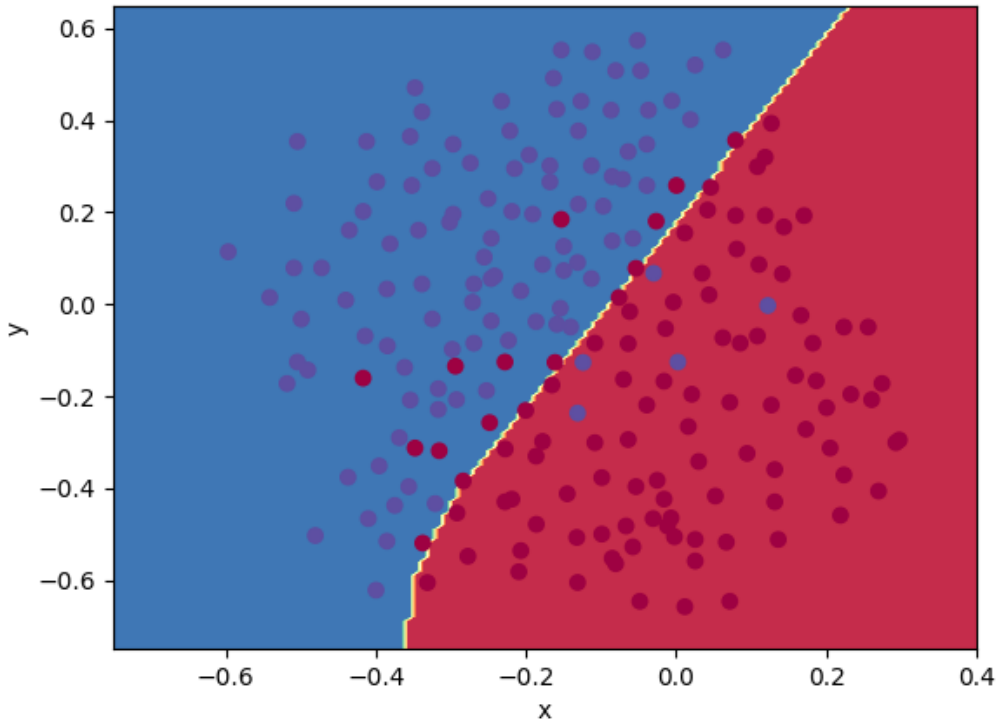


FIGURE 6.6 – Model With Dropout Designs Regularization - Figure 6.5's Graph

By this, we can see that preventing overfitting using BIBDs is possible, fast and efficient.

6.3.3 Model With Random Dropout Regularization

In this subsection, we will be sharing our simulation results of model that has random dropout method as regularization technique. That is, our 7-hidden-layer deep neural network's nodes will be turned off randomly on each iteration. First, lets share its accuracies on train and test datasets.

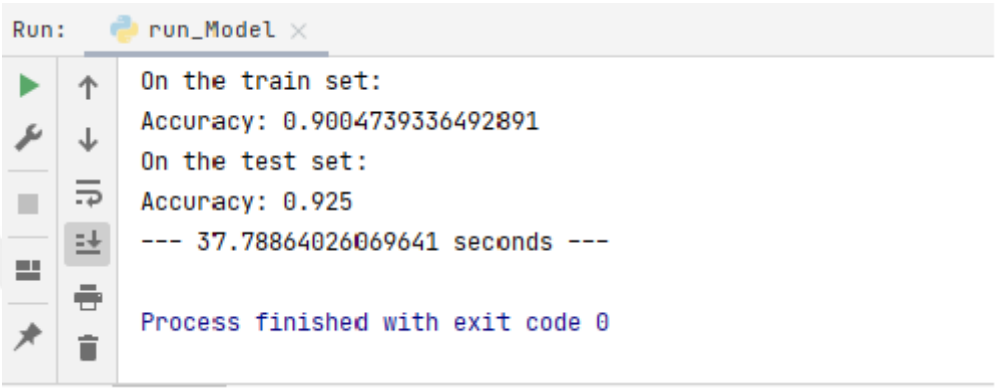


FIGURE 6.7 – Model With Random Dropout Regularization

We have 90% accuracy on train dataset where we have 92.5% accuracy on test dataset. We prevented overfitting using random method aswell but our success while using dropout designs was higher with 94%.

6.4 Implementations

In this section, we share the codes that we used to generate balanced incomplete block designs and to build our deep neural network.

6.4.1 Implementation of Algorithm X and BIBD Generation in C++

This section will cover `C++` codes of Algorithm X and balanced incomplete block design generation. We are writing generated block designs to a text file to read later in Python.

6.4.1.1 Solving Exact Cover Problem Using Algorithm X

We are solving the Exact Cover Problem using the following implementation of Algorithm X, which we explained detailed in Chapter 3. This implementation consists of two main items :

— Node structure and

— AlgorithmX class.

First, we are defining our fundamental structure N. This node structure has five components, D, C, U, R, L, that we explained detailed in Chapter 3. These components are public attributes of the structure N. D, C, U, R, L correspond to the N pointer `checkDown`, `ColumnNode`, `checkUp`, `checkRight` and `checkLeft` respectively. In addition to those attributes, N has integer attributes as `row_ID`, `column_ID`.

Second, we are defining necessary variables. The variable `list_header` is explained detailed in Chapter 3 and the variable `boolMat` is a matrix of size `MAX_ROW` $\times$ `MAX_COL` such that all of its entries are either 1 or 0. `MAX_ROW` and `MAX_COl` set up as 1000 in the beginning of the algorithm but we are simply using only the part we need of this 1000x1000 matrix. For example in (7,3,1), we do have 21 columns and 35 rows. Therefore, here, we are using only 21x35 of this 1000x1000 matrix. For larger BIBDs, the larger part of the matrix will be used and for BIBDs that we 'll require more space than 1000, this constant value should be enlarged.

Finally, we are defining AlgorithmX class, which is consisting of functions that are required while solving exact cover problem. This algorithm starts with its member function `findSolutions`. This function is taking a three integers starting point, number of rows of incidence matrix, number of columns of incidence matrix and a taking a 2-dimensional matrix, incidence matrix. We are generating this incidence matrix in the following section to give it as input to this function. The following is our `C++`: implementation.

```
#ifndef BLOCK_DESIGNS_V_ALGORITHMX_ALGORITHMX_H
```

```
#define BLOCK_DESIGNS_V_ALGORITHMX_ALGORITHMX_H
#define MAX_ROW 1000
#define MAX_COL 1000
#include <string>

struct N
{
    public:
        struct N *checkLeft;
        struct N *checkRight;
        struct N *checkUp;
        struct N *checkDown;
        struct N *ColumnNode;
        int row_ID;
        int column_ID;
};

static double SOLUTION_COUNT = 0;

static struct N *list_header = new N();
static bool boolMat[MAX_ROW][MAX_COL];
static struct N NodeMatrix[MAX_ROW][MAX_COL];
static std::vector<struct N *> solutions;

class AlgorithMX{
public:
static void fillBoolMat(std::vector<std::vector<int>> arrays) {
    std::vector<std::vector<int>>::iterator theRow;
    std::vector<int>::iterator theColumn;
    int r = 0;
    for (theRow = arrays.begin(); theRow != arrays.end(); theRow++) {
        r++;
        for (theColumn = theRow->begin(); theColumn != theRow->end(); theColumn++)
            boolMat[r][*theColumn - 1] = true;
        };
```

```
    };
};

static int bringRight(int j, int theNRow, int theNCol){
     return (j + 1) % theNCol;
}

static int bringLeft(int k, int theNRow, int theNCol)
{ return (k - 1 < 0) ? theNCol - 1 : k - 1; }

static int bringUp(int m, int theNRow, int theNCol)
{ return (m - 1 < 0) ? theNRow : m - 1; }

static int bringDown(int i, int theNRow, int theNCol)
{ return (i + 1) % (theNRow + 1); }

static void coverNode(struct N *targetNode)
{
struct N *theRow, *the_right_node;


struct N *colNode = targetNode->ColumnNode;

colNode->checkLeft->checkRight = colNode->checkRight;
colNode->checkRight->checkLeft = colNode->checkLeft;

for (theRow = colNode->checkDown; theRow != colNode; theRow = theRow->checkDown){
    for (the_right_node = theRow->checkRight; the_right_node != theRow;
    the_right_node = the_right_node->checkRight) {
        the_right_node->checkUp->checkDown = the_right_node->checkDown;
        the_right_node->checkDown->checkUp = the_right_node->checkUp;

        NodeMatrix[0][the_right_node->column_ID].the_node_count -= 1;
    }
```

```
}
}

static void uncoverNode(struct N *targetNode) {
    struct N *the_row_node, *the_left_node;

    struct N *colNode = targetNode->ColumnNode;

    for (the_row_node = colNode->checkUp; the_row_node != colNode;
        the_row_node = the_row_node->checkUp) {
        for (the_left_node = the_row_node->checkLeft;
            the_left_node != the_row_node;
            the_left_node = the_left_node->checkLeft)
        {
            the_left_node->checkUp->checkDown = the_left_node;
            the_left_node->checkDown->checkUp = the_left_node;

            NodeMatrix[0][the_left_node->column_ID].the_node_count += 1;
        }
    }

    colNode->checkLeft->checkRight = colNode;
    colNode->checkRight->checkLeft = colNode;
}

static N *get_the_minimum_column() {
    struct N *h = list_header;
    struct N *min_col = h->checkRight;
    h = h->checkRight->checkRight;
    do {
        if (h->the_node_count < min_col->the_node_count) {
            min_col = h;
        }
        h = h->checkRight;
    } while (h != list_header);
```

```
    return min_col;
};

static void search(int k) {
    struct N *the_row_node;
    struct N *the_right_node;
    struct N *the_left_node;
    struct N *ColumnNode;
    if (list_header->checkRight == list_header) {
        SOLUTION_COUNT++;
        write_solution_to_file(7, 3);
        return;
    }

    column = get_the_minimum_column();

    coverNode(column);

    for (the_row_node = column->checkDown; the_row_node != column;
    the_row_node = the_row_node->checkDown) {
        solutions.push_back(the_row_node);

        for (the_right_node = the_row_node->checkRight;
            the_right_node != the_row_node;
            the_right_node = the_right_node->checkRight)
        {
            coverNode(the_right_node);
        }


        search(k + 1);

        solutions.pop_back();
```

```
        column = the_row_node->ColumnNode;
        for (the_left_node = the_row_node->checkLeft; the_left_node != the_row_nod
        the_left_node = the_left_node->checkLeft)
        uncoverNode(the_left_node);
    }

    uncoverNode(column);
}

static N *makeToridolMatrix(int theNRow, int theNCol) {
for (int m = 0; m <= theNRow; m++) {
    for (int n = 0; n < theNCol; n++) {
        if (boolMat[m][j]) {
            int a, b;

            if (m) NodeMatrix[0][n].the_node_count += 1;

            NodeMatrix[m][n].ColumnNode = &NodeMatrix[0][n];

            NodeMatrix[m][n].row_ID = m;
            NodeMatrix[m][n].column_ID = n;



            a = m;
            b = n;
            do { b = bringLeft(b, theNRow, theNCol); } while (!boolMat[a][b] &&
             b != n);
            NodeMatrix[m][j].checkLeft = &NodeMatrix[m][b];

            a = m;
            b = n;
            do { b = bringRight(b, theNRow, theNCol); } while (!boolMat[a][b] &&
             b != n);
            NodeMatrix[m][n].checkRight = &NodeMatrix[m][b];
```

```
            a = m;
            b = n;
            do { a = bringUp(a, theNRow, theNCol); } while (!boolMat[a][b] &&
             a != m);
            NodeMatrix[m][n].checkUp = &NodeMatrix[a][n];

            a = m;
            b = n;
            do { a = bringDown(a, theNRow, theNCol); } while (!boolMat[a][b] &&
             a != m);
            NodeMatrix[m][n].checkDown = &NodeMatrix[a][n];
        }
    }
}

list_header->checkRight = &NodeMatrix[0][0];

list_header->checkLeft = &NodeMatrix[0][theNCol - 1];

NodeMatrix[0][0].checkLeft = list_header;
NodeMatrix[0][theNCol - 1].checkRight = list_header;
return list_header;
}

static void findSolutions(int starting_point, int nr, int nc,
                const std::vector<std::vector<int>> &given_array)
{
    int theNRow = nr;
    int theNCol = nc;


    for (int m = 0; m <= theNRow; m++) {
        for (int n = 0; n < theNCol; n++) {
            if (m == 0) boolMat[m][n] = true;
            else boolMat[m][n] = false;
```

```
        }
    }
    fillBoolMat(given_array);
    makeToridolMatrix(theNRow, theNCol);
    search(starting_point);
}

static void solve_design_problem(std::vector<int> vKlets) {
    int v = vKlets[0];
    int k = vKlets[1];
    std::vector<std::vector<int>> pairs = Combinations::combs(v, 2);
    std::vector<std::vector<int>> combs = Combinations::combs(v, k);
    std::vector<std::vector<int>> Inc_Matrix =
        IncidenceMatrix::getIncidenceMatrix(combs, pairs);
    std::vector<std::vector<int>> Design =
        IncidenceMatrix::getSets(Inc_Matrix);
    findSolutions(0, combs.size(), pairs.size(), Design);
}

static void write_solution_to_file(int v, int k)
{
    std::ofstream myFile;
    myFile.open(std::to_string(v)+"_"+std::to_string(k)+"_1_BIBDs.txt",
         std::ios_base::app);

    std::vector<std::vector<int>> combs = Combinations::combs(v,k);

    std::vector<struct N*>::iterator it;
    for(it = solutions.begin(); it!=solutions.end(); it++) {
        int val = (*it)->row_ID;
        myFile << val << " ";
    }
    myFile << "\n";
    myFile.close();
```

```
}

};
```

6.4.1.2 Incidence Matrix Generation

Following combination function is used in process of generating the incidence matrix that mentioned in Chapter 3.4.

```
#include <iostream>
#include "vector"
#include "algorithm"
#include "Combinations.h"


std::vector<std::vector<int>> Combinations::combs(int n, int r) {
    std::vector<bool> v(n);
    std::fill(v.end() - r, v.end(), true);
    std::vector<std::vector<int>> sol;
    do {
        std::vector<int> theRow;
        for (int i = 0; i < n; ++i) {
            if (v[i]) {
                theRow.push_back(i+1);
            }
        }
        sol.push_back(theRow);
    } while (std::next_permutation(v.begin(), v.end()));
    return sol;
}
```

And then, we generate the incidence matrix mentioned in Chapter (3.4) using the following :

```
#include "IncidenceMatrix.h"
```

```
#include "vector"
#include "iostream"
#include "algorithm"

bool IncidenceMatrix::subset_check(
    std::vector<int> subset_1,
    const std::vector<int> &subset_2
    ) {
    bool subset = true;
    for(const int & i: subset_2){
        if (!std::count(subset_1.begin(), subset_1.end(), i))
        subset = false;
    };
    return subset;
}

std::vector<std::vector<int>>
IncidenceMatrix::getIncidenceMatrix
(
    const std::vector<std::vector<int>> &klets,
    const std::vector<std::vector<int>> &columns
)
{
    std::vector<std::vector<int>> IncidenceArray;

    for(int i=0; i<klets.size(); i++){
        std::vector<int> temp_row;
        for(int j=0; j< columns.size(); j++){
            if(subset_check(klets[i],columns[j])){
                temp_row.push_back(true);
            }
            else{
                temp_row.push_back(false);
            }
        }
```

```
        IncidenceArray.push_back(temp_row);
    }

    return IncidenceArray;
}

std::vector<std::vector<int>>
IncidenceMatrix::getSets(std::vector<std::vector<int>> incidence_matrix)
{
    std::vector<std::vector<int>> sol;
    for(auto & i : incidence_matrix){
        std::vector<int> theRow;
        for(int j=0; j< i.size();j++){
            if(i[j]){
                theRow.push_back(j+1);
            }
        }
        sol.push_back(theRow);
    }
    return sol;
}
```

Finally, we call our source codes in the following main.cpp script :

```
#include <iostream>
#include "vector"
#include "Combinations.h"
#include "IncidenceMatrix.h"
#include "fstream"
#include "AlgorithmX.h"


int main() {
    std::vector<std::vector<int>> todo_list;
```

```
    /// Add values {v,k}s to todo_list to find {v,k,1}-designs.

    todo_list.push_back({7,3});

    // todo_list.push_back({13,4});

    for(auto& todo:todo_list){
        SOLUTION_COUNT = 0;
        AlgorithMX::solve_design_problem(todo);
        std::cout << SOLUTION_COUNT << " solutions found for "
            <<"("<<todo[0]<<","<<todo[1]<<",1)"<<" design."<<std::endl;
    }

    return 0;
}
```

After running main.cpp script, a text file that contains BIBDs will be created. We will use the generated BIBDs in next section.

6.4.2 Implementation of ML Model in Python

In this section, we share fundamental functions that we used while reading text file that contains balanced incomplete block designs and then using them in machine learning models. This process starts with running `C++` script to generate required .txt file then we read generated balanced incomplete block desings line by line in Python.

6.4.2.1 Construction of Model

In this section, we are sharing our neural network's model. The following neural network consist of seven hidden layers. As activation functions, RELU is used in the hidden layers and Sigmoid is used in the output layer. There are three forward propagation and backward propagation functions existing in the model. First one,

```
    forward_propagation_with_random_dropout,
```

refers to the forward propagation in neural network using random method. That is, we are dropping out the neurons in hidden layers randomly. Second one,

```
forward_propagation_with_dropout_designs,
```

is referring to the forward propagation process that uses BIBDs while dropping out neurons in the hidden layers. The third forward propagation function is referring to the usual forward propagation technique where we don't use any regularization method.

On the other hand, there are two backward propagation functions. First one,

```
backward_propagation_with_dropout,
```

refers to the backward propagation using dropout technique, other one refers to the usual technique. Note that, a seperate 3rd function for BIBDs wasn't needed here. Now lets look into the Model's definition :

```
import matplotlib.pyplot as pyplt

def Model(dataset_X, dataset_Y, model_parameters):
    layers_dims = model_parameters["layer_dimensions"]

    grads = {}
    costs = []

    parameters = initialize_parameters(layers_dims)

    for i in range(0, model_parameters["num_iters"]):
        if model_parameters["use_random_dropout"]:
            a_last, cache = forward_propagation_with_random_dropout(
                dataset_X, parameters, model_parameters)
        elif model_parameters["use_dropout_designs"]:
            a_last, cache = forward_propagation_with_dropout_designs(
                dataset_X, parameters, model_parameters)
        else:
```

```
            a_last, cache = usual_forward_propagation(
                dataset_X, parameters, model_parameters)

    cost = compute_cost(a_last, dataset_Y)

    #Backward propagation.
    if model_parameters["use_random_dropout"]:
        grads = backward_propagation_with_dropout(
            dataset_X, dataset_Y, cache, model_parameters)
    else:
        grads = usual_backward_propagation(
            dataset_X, dataset_Y, cache, model_parameters)

    #We update parameters by the following update parameters function.
    parameters = update_parameters(parameters,
        grads, model_parameters["learning_rate"])

    #We may print loss or costs depending on given model parameters.
    if model_parameters["print_cost"] and i %
     model_parameters["print_cost_per"] == 0:
        print("Cost after iteration {}: {}".format(i, cost))
    if model_parameters["print_cost"] and i %
     model_parameters["append_cost_per"] == 0:
        costs.append(cost)

    return parameters
```

6.4.2.2 Forward and Backward Propagations

In this section, we are sharing the forward and backward propagation functions that we called in our Model.

Forward Propagation with Dropout Designs

This function first takes weight and bias parameters. Then checks if we are applying dropout designs technique, if this is the case, we are getting the design with its isomorph permutations that we sent to the Model as parameter before starting it. We are randomly choosing one of the isomorph (7,3,1) balanced incomplete block designs and applying it to the matrix D called dropout matrix. D is of shape $(v, 211)$, i.e. $(7, 211)$. First, we are initializing D with all entries as 1. Then we are closing its entries depending on the BIBD we have randomly chosen. For example, if our chosen BIBD's first block is $\{1, 2, 4\}$, then we are closing 1st, 2nd and 4th rows of matrix D. In other words, we are setting $D[0]$, $D[1]$ and $D[3]$ to the 0.

After being done with dropout matrix D of each layer for 7 hidden layers, we are calculating U and A by $WX + b$ and $\alpha(U)$ respectively where α is the activation function we are using on that layer. If we are in a hidden layer, we are multiplying A with D entrywise to dropout our values. If dropout designs is not the case, this function simply calculates U and A functions without multiplying it entry wise with D.

```
def forward_propagation_with_dropout_designs(X, parameters, model_params):
    np.random.seed(model_params["random_seed"])
    methods = model_params["activation_functions_in_order"]
    params = dict()
    for i in range(1, len(methods) + 1):
        params["W" + str(i)] = parameters["W" + str(i)]
        params["b" + str(i)] = parameters["b" + str(i)]
    dropout_exist = model_params["use_dropout_designs"] or model_params[
    "use_random_dropout"]

    if model_params["use_dropout_designs"]:
        design = model_params["design"]
        isomorph_incidence_matrices = design.isomorph_designs_sack
        d = random.choice(isomorph_incidence_matrices)
        for i in range(1, len(methods)):
            params["D" + str(i)] = np.ones((design.v, X.shape[1]), dtype=int)
        for i, block in enumerate(d):
            for theRow in block:
                params["D" + str(i + 1)][theRow-1] = 0
```

```
    for n, m in enumerate(methods):
        if m == "relu":
            if n == 0:
                params["U" + str(n + 1)] = np.dot(params["W" + str(n + 1)],
                 X) + params["b" + str(n + 1)]
                params["A" + str(n + 1)] = relu(params["U" + str(n + 1)])
            else:
                params["U" + str(n + 1)] = np.dot(params["W" + str(n + 1)],
                     params["A" + str(n)]) + params["b" + str(n + 1)]
                params["A" + str(n + 1)] = relu(params["U" + str(n + 1)])

            if dropout_exist and n != len(methods) - 1:
                params["A" + str(n + 1)] = params["A" + str(n + 1)] *
                 params["D" + str(n + 1)]
                params["A" + str(n + 1)] = params["A" + str(n + 1)] /
                 model_params["keep_prob"]
            elif m == "sigmoid" and n == len(methods) - 1:
            if n == 0:
                params["U" + str(n + 1)] = np.dot(params["W" + str(n + 1)],
                 X) + params["b" + str(n + 1)]
                params["A" + str(n + 1)] = sigmoid(params["U" + str(n + 1)])
            else:
                params["U" + str(n + 1)] = np.dot(params["W" + str(n + 1)],
                 params["A" + str(n)]) + params[
                  "b" + str(n + 1)]
                params["A" + str(n + 1)] = sigmoid(params["U" + str(n + 1)])

    cache = dict()
    for key in params.keys():
        cache[key] = params[key]

    return params["A" + str(len(methods))], cache
```

Forward Propagation with Random Dropout

The following is the forward propagation function with random dropout method.

```
def forward_propagation_with_random_dropout(X, parameters, model_params):
    np.random.seed(1)
    methods = model_params["activation_functions_in_order"]
    params = dict()
    for i in range(1, len(methods) + 1):
        params["W" + str(i)] = parameters["W" + str(i)]
        params["b" + str(i)] = parameters["b" + str(i)]

    for i, m in enumerate(methods):
        if m == "relu":
            if i == 0:
                params["U" + str(i + 1)] = np.dot(params["W" + str(i + 1)], X) + p
                params["A" + str(i + 1)] = relu(params["U" + str(i + 1)])
            else:
                params["U" + str(i + 1)] = np.dot(params["W" + str(i + 1)], params
                "b" + str(i + 1)]
                params["A" + str(i + 1)] = relu(params["U" + str(i + 1)])

            if dropout_exist and i != len(methods) - 1:
                params["D" + str(i + 1)] = np.random.rand(params["A" + str(i + 1)]
                params["A" + str(i + 1)].shape[1])
                params["D" + str(i + 1)] = (params["D" + str(i + 1)] < model_param
                params["A" + str(i + 1)] = params["A" + str(i + 1)] * params["D" +
                params["A" + str(i + 1)] = params["A" + str(i + 1)] / model_params
        elif m == "sigmoid" and i == len(methods) - 1:
            if i == 0:
                params["U" + str(i + 1)] = np.dot(params["W" + str(i + 1)], X) + p
                params["A" + str(i + 1)] = sigmoid(params["U" + str(i + 1)])
            else:
                params["U" + str(i + 1)] = np.dot(params["W" + str(i + 1)], params
                "b" + str(i + 1)]
                params["A" + str(i + 1)] = sigmoid(params["U" + str(i + 1)])
```

```
    cache = dict()
    for key in params.keys():
        cache[key] = params[key]

    return params["A" + str(len(methods))], cache
```

Backward Propagation With Dropout

In backward propagation functions, we are taking the ghosted (or non-ghosted if dropout technique is not used) neural networks and calculating gradient parameters. In addition to the usual backward propagation function, if dropout technique is being used, then we are multiplying matrix dA with previously obtained dropout matrix D.

```
def backward_propagation_with_dropout_designs(X, Y, cache, model_params):
    methods = model_params["activation_functions_in_order"]
    gradients = dict()
    s = X.shape[1]
    for n in range(len(methods), 0, -1):
        if n == len(methods):
            gradients["dU" + str(n)] = cache["A" + str(n)] - Y
            gradients["dW" + str(n)] = 1. / s * np.dot(gradients[
             "dU" + str(n)], cache["A" + str(n - 1)].T)
            gradients["db" + str(n)] = 1. / s * np.sum(gradients[
             "dU" + str(n)], axis=1, keepdims=True)
        elif len(methods) > n > 1:
            gradients["dA" + str(n)] = np.dot(cache["W" + str(n + 1)].T,
             gradients["dU" + str(n + 1)])
            gradients["dA" + str(n)] = gradnents["dA" + str(n)] *
             cache["D" + str(n)]
            gradients["dA" + str(n)] = gradients["dA" + str(n)] /
             model_params["keep_prob"]
            gradients["dU" + str(n)] = np.multiply(gradients["dA" +
             str(n)], np.int64(cache["A" + str(n)] > 0))
            gradients["dW" + str(n)] = 1. / s * np.dot(gradients[
```

```
            "dU" + str(n)], cache["A" + str(n - 1)].T)
            gradients["db" + str(n)] = 1. / s * np.sum(gradients[
            "dU" + str(n)], axis=1, keepdims=True)
        elif i == 1:
            gradients["dA" + str(n)] = np.dot(cache["W" + str(n + 1)].T,
             gradients["dU" + str(n + 1)])
            gradients["dA" + str(n)] = gradients["dA" + str(n)] *
             cache["D" + str(n)]
            gradients["dA" + str(n)] = gradients["dA" + str(n)] /
             model_params["keep_prob"]
            gradients["dU" + str(n)] = np.multiply(gradients["dA" +
             str(n)], np.int64(cache["A" + str(n)] > 0))
            gradients["dW" + str(n)] = 1. / s * np.dot(gradients[
             "dU" + str(n)], X.T)
            gradients["db" + str(n)] = 1. / s * np.sum(gradients[
             "dU" + str(n)], axis=1, keepdims=True)
    return gradients
```

Usual Backward Propagation

The following is the usual backward propagation function where we don't apply any regularization technique.

```
def backward_propagation(X, Y, cache, model_params):
    random.seed(model_params["random_seed"])
    methods = model_params["activation_functions_in_order"]
    gradients = dict()
    s = X.shape[1]
    for i in range(len(methods), 0, -1):
        if i == len(methods):
            gradients["dU" + str(i)] = cache["A" + str(i)] - Y
            gradients["dW" + str(i)] = 1. / s * np.dot(gradients["dU" + str(i)], c
            gradients["db" + str(i)] = 1. / s * np.sum(gradients["dU" + str(i)], a
        elif len(methods) > i > 1:
            gradients["dA" + str(i)] = np.dot(cache["W" + str(i + 1)].T, gradients
```

```
            gradients["dU" + str(i)] = np.multiply(gradients["dA" + str(i)], np.in
            gradients["dW" + str(i)] = 1. / s * np.dot(gradients["dU" + str(i)], c
            gradients["db" + str(i)] = 1. / s * np.sum(gradients["dU" + str(i)], a
        elif i == 1:
            gradients["dA" + str(i)] = np.dot(cache["W" + str(i + 1)].T, gradients
            gradients["dU" + str(i)] = np.multiply(gradients["dA" + str(i)], np.in
            gradients["dW" + str(i)] = 1. / s * np.dot(gradients["dU" + str(i)], X
            gradients["db" + str(i)] = 1. / s * np.sum(gradients["dU" + str(i)], a

    return gradients
```

6.4.2.3 Dropout Designs Method

In this section, we share code of closing nodes in neural network and then we explain the parameters we use in our model.

```
dropout_exist = model_params["use_dropout_designs"] or
    model_params["use_random_dropout"]
if model_params["use_dropout_designs"]:
    design = model_params["design"]
    isomorph_incidence_matrices = design.isomorph_designs_sack
    d = random.choice(isomorph_incidence_matrices)
    for i in range(1, len(methods)):
        params["D" + str(i)] = np.ones((design.v, X.shape[1]),
         dtype=int)
    for i, block in enumerate(d):
        for theRow in block:
            params["D" + str(i + 1)][theRow-1] = 0
```

where modelparams=

```
    model_parameters = {
        "num_iters": 30000,
        "keep_prob": 0.57,
        "learning_rate": 0.05,
```

```
    "design": _7_3_1_Design,
    "random_seed": random_seed,
    "activation_functions_in_order": ["relu","relu","relu","relu",
        "relu","relu","relu", "sigmoid"],
    "layer_dimensions": [train_X.shape[0], 7, 7, 7, 7, 7, 7, 7, 1],
    "use_dropout_designs": True,
    "use_random_dropout": False,
    "print_cost": True,
    "print_cost_per": 10000,
    "append_cost_per": 1000,
}
```

is a Python dictionary that holds key features of Model and given to the Model at the start of training. *numiters* is an integer that determines number of iterations, *keepprob* is used to scale weights of neurons in neural network after dropping them out using dropout designs technique or random dropout technique. If we are applying random dropout technique, in addition to this, *keepprob* will be the percentage of how many neurons in the neural network will be kept. That is, if $keepprob = 0.57$ in random dropout, this means we are keeping 57% of neurons of hidden layers in our deep neural network. This is not the case while using dropout designs technique.

learning rate is our learning rate that we use to update our parameters W and b. *design* is a Python class defined as :

```
class IterDesign:
    def __init__(self, v, k, lamda):
        self.v = v
        self.k = k
        self.lamda = lamda
        self.design = None
        self.isomorph_designs_sack = []
```

where $self.design$ is randomly picked BIBD over other generated designs and *isomorph designs sack* is list of permutations of randomly picked BIBD in $self.design$. In (7,3,1) case, *isomorph designs sack* consist of 7!=5040 elements. Before starting to the training

of model, we are sampling 2.000 (7,3,1) designs over 5040 randomly with

```
sample_incidence_matrices_indices = random.sample(iterations, 2000).
```

In dropout designs techinuqe, on each iteration, we choose one of those 2000 sampled designs randomly and apply it to the neural network. *randomseed* is determined seed for random functions to obtain stable results. *activation functions in order* is the list of activation functions that are going to be used in each hidden layer and in the output layer. *layer dimensions* are the sizes of each layer in neural network.

7 CONCLUSION

We started by giving brief introduction of balanced incomplete block designs. Then we deep dived into the Algorithm X and Exact Cover Problems. We used these definitions and algorithms on our machine learning models that we built later.

If we come to the comparison of techniques we applied, all calculation results given in chapter 6 was done for same number of iterations, but as it can be seen from figures, dropout designs technique finished earlier than random dropout technique and gave better accuracy on test dataset. By this, it is possible to claim that using dropout technique with BIBDs is as efficient as using random method. On the other hand, implementing random method is a lot easier than implementing dropout designs method. Because, random method can be applied to almost all deep neural network models with ease but to apply BIBD to a model, model must be constructed thinking selected BIBD.

In our simulations, we applied those methods to the deep neural network with 7 hidden layers specifically because we wanted to use (7,3,1)-BIBDs. In order to apply a different BIBD to the model, we would need to change model's hidden layer count with layers sizes (or change the way we train model). The difficulty and slowness of using large networks makes it hard to avoid overfitting by pooling the predictions of many different large neural networks at test time [17].

As a conclusion, building the correct model for your input size, regularization technique, training size, test size, learning rate and number of iterations is essential to have a good accuracy on predictions after training our model. Therefore dropout designs technique might be inefficient where researchers want quick solutions.

REFERENCES

[1] Michael A. Nielsen. Neural networks and deep learning, 2018.

[2] Charles J. Colbourn and Paul C. van Oorschot. Applications of combinatorial designs in computer science. *ACM Comput. Surv.*, 21(2) :223–250, June 1989.

[3] Ian Anderson. *Combinatorial Designs and Tournaments, Oxford Lecture Series in Mathematics and Its Applications*. OXFORD University Press, 1997.

[4] E. F. Assmus. *Designs and Their Codes*. Cambridge University Press, USA, 1992.

[5] Douglas R. Stinson. *Combinatorial Designs : Constructions and Analysis*. SpringerVerlag, 2003.

[6] S. S. Shrikhande and D. Raghavarao. A method of construction of incomplete block designs. *Sankhyā : The Indian Journal of Statistics, Series A (1961-2002)*, 25(4) :399–402, 1963.

[7] Donald E. Knuth. Dancing links. *Millenial Perspectives in Computer Science, 2000, 187–214*, (Knuth migration 11/2004), November 2000.

[8] Hirosi Hitotumatu and K. Noshita. A technique for implementing backtrack algorithms and its application. *Inf. Process. Lett.*, 8 :174–175, 1979.

[9] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972.

[10] Doubly linked list, Apr 2021. https://en.wikipedia.org/wiki/Doubly_linked_list.

[11] Geoffrey E. Hinton, Nitish Srivastava, A. Krizhevsky, Ilya Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv*, abs/1207.0580, 2012.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Shoko Chisaki, Ryoh Fuji-Hara, and Nobuko Miyamoto. Combinatorial designs for deep learning. *Journal of Combinatorial Designs*, 28(9) :633–657, May 2020.

[14] Neural networks and deep learning coursera. `https://www.coursera.org/learn/neural-networks-deep-learning`.

[15] Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs, Second Edition (Discrete Mathematics and Its Applications)*. Chapman, Hall/CRC, 2006.

[16] P. J. Cameron and J. H. van Lint. *Designs, Graphs, Codes and their Links*. London Mathematical Society Student Texts. Cambridge University Press, 1991.

[17] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15 :1929–1958, 06 2014.

BIOGRAPHICAL SKETCH

My name is Muhammet Ali Öztürk. I finished my bachelor's at Galatasaray University mathematics department with top degree in 2019 and enrolled a master program there the same year. While continuing my master's, I am working as an associate research scientist in AI, R&D Department at Afiniti. Afiniti is the world's leading applied artificial intelligence provider that transforms how individuals interact with each other based on their behavior, and has delivered more than $4 billion in revenue to its customers to date.