Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Kertész Gergő Csaba

KVANTUM ALAPÚ VÉLETLENSZÁMOK TESZTELÉSE NEURÁLIS HÁLÓK SEGÍTSÉGÉVEL

KONZULENS

Dr. Bacsárdi László

BUDAPEST, 2018

Tartalomjegyzék

Összefoglaló

Manapság egyre elterjedtebb a számítástudomány világában a kvantumosság. Egyre több területen alkalmazzák nagy sikerrel és várakozásokat felülmúló teljesítménnyel. Ilyen területek közé tartozik például a kvantumprocesszor, a kvantum vezetékes és műholdas kommunikáció és a kvantum véletlenszám-generálás is.

A véletlenszám generálás a kriptográfia és ezáltal a biztonságos kommunikáció szempontjából alapvető igény, ezért nagyon fontos jó minőségű véletlenszámokat generálni. Itt fekszik a terület egyik legnagyobb kihívása, mivel a véletlenszámokat nagyon nehéz minősíteni, többek között azért is, mert nehéz elkülöníteni egy véletlenszerűnek tűnő mintát egy valóban véletlentől, vagy egy mintát egy mintának tűnő véletlen ismétlődéstől.

A kvantum alapú véletlenszám-generátorra többféle architektúra is létezik, melyek kvantumos jelenségekre alapozva, megbízhatóbb random számokat generálnak mint a legtöbb klasszikus. Ilyen architektúrák például az útválasztáson, beérkezési időn vagy fotonszámlálásan alapulnak.

Ezen dolgozat témája a kvantum véletlenszám-generáláshoz kapcsolódik, azon belül is a véletlenség tesztelés egy nem szokványos módjához, a neurális hálózatokkal való teszteléshez. A neurális hálózatok a mai informatika egy másik úttörő témája, melynek felhasználási területei egyre csak növekszenek. A neurális hálózatok elterjedésének egyik legfőbb oka a képességük a minta- és képfelismerésre és az úgynevezett klaszterizálásra.

A TDK-dolgozatom is a hálózatok ezen képességeire alapszik. Munkámban kidolgozásra került néhány neurális architektúra a véletlenszámok tesztelésére. Az első egy egyszerű többréteges perceptron architektúra, amely a generátorok on-line tesztelésére alkalmas, a második egy konvolúciós háló, amely bitképek alapján próbál különbséget tenni igazi (klasszikus) véletlenszám, kvantumos véletlenszám és álvéletlenszám között. A harmadik pedig N-bites véletlenszámokat próbál meg klaszterezni. A kihívás ezekben a helyzetekben csak részben adódik a neurális hálókból, a nagyobb része a feladatnak a teszt által mutatott eredmények helyes értelmezése.

Abstract

Quantum technology is a rapidly expanding field of information technology. It's been successfully applied to numerous fields with expectation surpassing efficiency. These fields include: quantum processors, quantum communications via satellite or cable and quantum random number generation.

Random Number Generators (RNG) are an essential part of cryptography and such, also of secure communication, and so, generating random numbers of good quality is of utmost importance. However, a big question of RNG's is exactly the following: What counts as a random number of good quality? For example, how can you say if a random-appearing number is truly random or just the result of a complex algorithm or if a pattern is truly a pattern or just a coincidence in a 10^{30} bit random number

Multiple architectures exist for quantum based RNG's (QRNG), which, based on quantum phenomena, are more reliable than most of the classic architectures. Such architectures are based on, for example, path selection, arrival time and photon counting.

The main topic of this text is Quantum RNG (QRNG), and more precisely, random number testing. The testing is done in a non-regular manner, with the help of neural networks. Neural networks are an another hot topic of informatics, it's field of application quickly growing. The main reason for the popularity of said networks is their ability of pattern and image recognition and clusterization.

My work is based on the aforementioned capabilities of neural networks. I worked out some neural architectures for random number testing. The first is a simple Multi-Layer Perceptron (MLP) architecture, which can be used for on-line testing of RNG's. The second is a Convolutional Neural Network (CNN), which tries to differentiate between True RNG's, Pseudo RNG's and Quantum RNG's. The third is a Kohonen network, or Self-Organizing Map (SOM), which tries to cluster N-bit random numbers. The challenge of the problem is only partly the construction of said architectures. A more difficult part is the proper analysis of the results, and drawing the correct conclusions from them.

1 Bevezető

1.1 A kvantummechanikáról röviden

Az absztraktban említettek szerint a kvantumos véletlenszám-generátorokkal is foglalkoztam. Ahhoz, hogy a kvantum alapú véletlenszám generálást bemutathassam, először szeretném ismertetni a kapcsolódó elméletet, amit ehhez tudni kell, mivel a kvantuminformatika alapjai még nem széles körben ismertek. A kvantummechanikát csak futólag mutatom be, de fontos az alapok megértéséhez és említve lesz a későbbiekben olyan dolgok leírásánál, amelyek fontosak a végső munkám szempontjából. A következőekben bemutatott elméleti alapnak egy, a BME munkatársai által alkotott, angol nyelvű könyv [1] az információforrása.

Mint a legtöbb matematikai és fizikai modellnek, a kvantummechanikának is vannak feltevései/axiómái. Ezeket posztulátumnak hívjuk, és négy van belőlük, amelyek a kapcsolódó elmélet alapját adják. Kutatások szerint az univerzum legtöbb szabálya és törvénye visszavezethető erre a négy posztulátumra (néhány kivétellel).

A posztulátumok:

1. Posztulátum : Állapottér.

   Az aktuális állapota bármilyen zárt fizikai rendszernek leírható egy állapotvektorral ***(v)****, mely komplex együtthatókkal rendelkezik és egység hosszú egy Hilbert-térben*[1] *(V).*

2. Posztulátum: Evolúció.

   Bármely zárt fizikai rendszer időbeli evolúciója karakterizálható uniter[2] *transzformációk által melyek csak az evolúció kezdő és záró időpontjától függnek.*

3. Posztulátum: Mérés.

---

[1] Hilbert-tér: Komplex, lineáris állapottér mely rendelkezik belső szorzattal (skalárszorzattal)

[2] Uniter operátor: Egy operátor uniter, ha az operátor mátrixának adjungáltja megegyezik az inverzével. Adjungált : A mátrix transzponáltjának a komplex konjugáltja.

Komplex konjugált: A mátrix minden elemének vesszük a konjugáltját.

Bármely kvantum mérés leírható mérésoperátorok egy halmaza { M_m } által, ahol m a mérések lehetséges eredményeit reprezentálja. A valószínűsége egy adott m eredmény mérésének, ha tudjuk, hogy a rendszer $\mathbf{v}$ *állapotban van a következőképpen számítható ki:*

$$\mathrm{P}(m \mid \mathbf{v}) = \mathbf{v}^{\dagger}\mathrm{M}_m^{\dagger}\mathrm{M}_m\mathbf{v}$$

m mérése után a rendszer pedig a $\mathbf{v}'$ *állapotba megy át:*

$$\mathbf{v}' = \frac{\mathrm{M}_m\mathbf{v}}{\sqrt{\mathbf{v}^{\dagger}\mathrm{M}_m^{\dagger}\mathrm{M}_m\mathbf{v}}}$$

Ez azért van mert a klasszikus valószínűségi elmélet megköveteli, hogy:

$$\sum_m \mathrm{P}(\, m \mid \mathbf{v}\,) = \sum_m \mathbf{v}^{\dagger}\mathrm{M}_m^{\dagger}\mathrm{M}_m\mathbf{v} \equiv 1$$

a mérési operátorok teljesítsék a teljességi relációt:

$$\sum_m \mathrm{M}_{\mathrm{m}}^{\dagger}\mathrm{M}_{\mathrm{m}} \equiv 1$$

4. Posztulátum: Kompozit(Összetett) rendszer.

Egy kompozit fizikai rendszer (W) kiszámítható az őt előállító egyedi rendszerek tenzorszorzatával $W = V \otimes Y$. *Továbbá, ha* $\boldsymbol{v} \in V$ *és* $\boldsymbol{y} \in Y$ *akkor az együttes állapota a kompozit rendszernek:* $\boldsymbol{w} = \boldsymbol{v} \otimes \boldsymbol{y}$.

Talán a legfontosabb posztulátum a dolgozatom szempontjából a harmadik, mivel a véletlenszám generálásának a folyamatát nagyban befolyásolja az eredmények kinyerése, amivel egy megfigyelést iktatunk be a rendszerbe, így változtatjuk az állapotát.

A mérések nem reverzibilis cselekvések, ezért kivételt képeznek a második posztulátum alól. Ugyanakkor pont emiatt a tulajdonság miatt képesek a kvantum és a klasszikus világ „összekötésére", általuk nyerhetünk bepillantást a kvantumvilágba. Hátránya ennek a tulajdonságnak azonban, hogy sosem tudjuk pontosan, mit mértünk, ugyanis mivel az operátorok nem reverzibilisek, ezért nem számíthatjuk ki, milyen állapotban volt a rendszer a megfigyelésünk előtt abból, hogy a mérés milyen eredményt mutatott és milyen állapotba jutott utána, csak egy valószínűségi becslést adhatunk rá. Ezért is nehéz a kvantumvilágban dolgozni, egy algoritmus,

generátor megalkotása egy dolog, abból eredményt kinyerni, amire biztosan mondhatjuk, hogy az amit kapni, akartunk egy másik.

1.2 Kvantuminformatika

A fő különbség a klasszikus és a kvantuminformatika között az alapegységek között a legfeltűnőbb: a bit és a kvantumbit (angolul quantum bit azaz qubit) valamint a logikai kapuk és a kvantumkapuk között. Ezekből építkezni legtöbbször már fizikailag hasonlóan lehet eltérő súlyosságú különbségekkel, például a kvantumáramkörökben optikai kábeleket használnak főleg, de ezt használják a klasszikus informatikában is. A különbség az áramkörök felépítésben a logika miatt van, amit a kvantumbitek és a kvantum kapuk működése okoz.

1.2.1 Kvantumbitek

A klasszikus bit egy nem túl komplex fogalom: valami, aminek két elkülöníthető állapota van, gyakran 0-val és 1-el kódolva és egyszerre csak egy állapotban lehet. Például van áram (1) ←→ nincs áram (0) vagy egy érme feldobásának eredmény, fej vagy írás.

A kvantumbitek ennél már komplikáltabbak. Az első posztulátumnak megfelelően egy zárt rendszernek tekinthető kvantumbit egy Hilbert-térben létező, komplex együtthatójú és egységnyű hosszú állapot vektor [1]. A vektornak a komplex együtthatói a kvantumbitnek a „számítási bázisvektor”-ait súlyozzák. A számítási bázisvektorok általában a következőek: $|0\rangle$ és $|1\rangle$. Tehát a kvantumbit egy általános állapotvektora a következőként néz ki:

$$|\varphi\rangle = a|0\rangle + b|1\rangle\,, \qquad (1)$$

ahol $a^2 + b^2 = 1$ az első posztulátumnak megfelelően. Így egy általános $|\varphi\rangle$ állapotvektora a qubitnek nem más mint a lineáris kombinációja a két számítási bázisvektornak (a Hilbert-tér rendszer egységnyi hosszú bázisvektorainak). A komplex együtthatók az úgynevezett valószínűségi amplitúdók, így teljesíteniük kell az $|a|^2+|b|^2 = 1$ összefüggést. Ezt a modellalkotásnál már kielégítettük az első posztulátumnak hála és így a harmadik posztulátummal is harmóniában maradtunk és a harmadik értelmében a számok jelentése a következő: mérés $|0\rangle$-át $|a|^2$ míg $|1\rangle$-et $|b|^2$ valószínűséggel eredményez. Mindenképpen hangsúlyozni kell a különbséget a klasszikus bittel szemben. A klasszikus bit a mérés előtt,alatt és után is egy állapotban volt, a mérés nem változtatott rajta. Viszont a mérés előtt a kvantumbit valamilyen mértékben mindkét állapotot tartalmazta a $|\varphi\rangle$ állapotvektorában, és csak a mérés

miatt „csapódott” le az egyik vagy másik állapotba. Kvantumbitre példák: egy elektronnál a spin-up és spin-down állapot, egy fotonnál pedig a horizontális vagy vertikális polarizáció.

1.2.2 Kvantumregiszterek

Ahogy a klasszikus informatikában, a kvantuminformatikában is készíthetünk kvantumregisztereket kvantumbitekből. Egy n-qubites kvantumregiszter tartalmazhatja bármelyiket az $N = 2^n$ dimenziójú számítási bázisvektorból, vagy egy tetszőleges szuperpozícióját ezeknek. Ha két kvantumbitet összekapcsolunk, akkor két zárt rendszerből teszünk össze egy új, összetett rendszert, aminek megalkotásához segítségül hívhatjuk a negyedik posztulátumot. Azaz azt, hogy a kompozit rendszer állapota az őt alkotó állapotok tenzor szorzata.

Vegyük példának a következő két kvantumbitet:

$$|\varphi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

$$|\varphi_2\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

$$|\varphi\rangle = |\varphi_1\rangle|\varphi_2\rangle = |\varphi_1, \varphi_2\rangle = |\varphi_1\varphi_2\rangle = \frac{|0\rangle\otimes|0\rangle - |1\rangle\otimes|0\rangle + |0\rangle\otimes|1\rangle - |1\rangle\otimes|1\rangle}{2} = \frac{|00\rangle - |10\rangle + |01\rangle - |11\rangle}{2}$$

Tehát így egy két kvantumbites regisztert kapunk, melynek az állapota négy lineárisan súlyozott számítási bázisvektorból áll. Ezek a klasszikus két bites regiszterek lehetséges tartalmai, a különbség, hogy a kvantumos esetben mind a négy állapot jelen van egy darab kvantumregiszterben.

Ha megmérjük az első bitet, akkor láthatóvá válik a harmadik posztulátum fontossága: tegyük fel, hogy megmértük $|\varphi_1\rangle$-t és az eredmény $|0\rangle$, ekkor a kvantumregiszter tartalma „megváltozik”.

$$|\varphi\rangle = |0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{|00\rangle - |01\rangle}{\sqrt{2}}$$

A kimaradó számítási vektorokat is vehetjük úgy, hogy benne vannak, csak nulla valószínűségi amplitúdóval.

Egy n kvantumbites kvantumregiszter általános állapota:

$$|\varphi\rangle = \sum_{i=0}^{2^n-1} \varphi_i |i\rangle,$$

ahol φ_i a valószínűségi amplitúdója az i-edik számítási bázisvektornak. Amit a következőt jelenti: tgy kvantumregiszter egyszerre 2^n klasszikus számot tartalmaz magában. Ez nagyon nagy számítási potenciált jelent és nagyon nagy kiszámíthatatlanságot ami véletlenszám generálásnál nagyon hasznos.

1.2.3 Kvantuminformatika mai állása

Napjainkban a kvantuminformatika egy egyre jobban elterjedt területe az informatikának, egyetemek és cégek kezdenek el foglalkozni vele, üzleti, biztonsági, tanulmányi célból is. Kínában 2016-ban Föld körüli pályára állították az első műholdat, mely hosszútávú, kvantum kulcsszétosztással biztosított kommunikációs hálózatot létesít földi csomópontokkal [2], a kanadai D-Wave nevű cég pedig már több kvantumszámítógépet is piacra vitt és adott el [3]. Egyetemek közül a BME-nek is több kvantum alapú kommunikációs berendezése van a HIT Mobil Kommunikációs és Kvantumtechnológiák Laboratóriumban[4], az ausztrál ANU kvantum véletlenszám-generátoráról pedig neten keresztül is lehet véletlenszámokat lekérni[5].

1.3 Neurális hálózatokról röviden

A neurális hálózatok téma iránti érdeklődése az embereknek folyamatosan változik, amikor elsőnek megjelentek az informatika színterén, mindenkit nagyon érdekeltek, majd téves korlátokat állítottak nekik neves emberek a mesterséges intelligencia területén (Marvin Minsky, Seymour Papert) egy publikációban [6] és az érdeklődés elhalt. Néhány évtizeddel később bebizonyították, hogy ezek a korlátok tévesek és átléphetőek és mára már megint fellendülőben van a téma iránti érdeklődés és nagyon sok helyen használnak neurális hálózatokat.

Az első neurális hálózat egy egyszerű perceptron (a Rosenblatt Perceptron) volt, ami az emberi agyban lévő neuronok felfedezését követte és azt modellezte [7]. Ahogy fejlődött a terület, az újfajta hálózatok közül is van olyan, ami az emberi biológiát tekinti alapnak, például a konvolúciós neurális hálózat, ami az agyunkban lévő képfelismerő sejteket modellezi [8][9].

2 Modellalkotás előtt

2.1 NIST Testing Suite

A NIST Testing Suite dokumentációnak olvasása által ismerkedtem meg és mélyültem el a véletlenszámok világában [10]. Az olvasás által fogalmaztam meg a gondolataim a véletlenszámokról, hogyan is lehetne őket tesztelni, és mik a problémák velük. A dokumentációból szeretnék kiemelni néhány részletet, és leírni, miért éreztem jó iránynak a neurális hálózatokkal való tesztelést.

2.1.1 Néhány alapelv

- Egyenletesség: Bármely pontján egy véletlen vagy pszeudo véletlen bitszekvencia generálásnak, a 0 előfordulásának a valószínűsége legyen egyenlő az 1-évl
- Skálázhatóság: Egy teszt, ami alkalmazható egy szekvenciára, legyen alkalmazható annak részszekvenciájára is. Hogyha a szekvencia véletlenszerű, akkor a résszekvenciának is véletlenszerűnek kell lennie. Tehát egy olyan szekvenciának, ami átment a teszten, bármelyik résszekvenciáját vesszük, annak is át kell mennie a teszten.
- Konzisztencia: Egy generátor viselkedésének konzisztensnek kell lenni a kezdő értékek felett. Nem elég egy pszeudo vélelenszám-generátort (PRNG) egy seed-en tesztelni, vagy egy igazi véletlenszám-generátort (TRNG) egy fizikai kimeneten.

Ezeket az elveket a munkám során igyekeztem betartani.

2.1.2 A tesztekről néhány szó

A NIST Testing Suite 15 darab statisztikai tesztet tartalmaz véletlenszámok tesztelésére. A tesztek leírásait olvasva gondolkoztam azon, hogy ezek a tesztek egy véletlenszám-generátornak vajon milyen hibáit nem fognák-e meg. Mivel a legtöbb teszt szigorúan statisztikai, ezért viszonylag kevés foglalkozik a 15-ből mintákkal, a legtöbb az eloszlásfüggvényt közelíti. Az Overlapping Template Matching Test például módosítható minta keresésre, a spektrum tesztet pedig módosítani sem kell, hogy ezt csinálja, ezek viszont csak periodikus mintákat tudnak felismerni, ekkor jutottak eszembe a neurális hálózatok a mintafelismerési képességükkel először.

3 Modellhez szükséges adathalmazok

3.1 Adatformátum

A teszteléshez adatok kellenek, az adatok pedig különböző formátumokban jelennek meg a világban, és magas szinten sokféleképpen ábrázolhatóak számítógépen is. A véletlenszám generálás típusonként másféleképpen történik. Pszeudo véletlenszámok esetén egy bonyolult függvény kimenete valamilyen seed-el. Igazi véletlenszám esetén egy fizikai jelenség érzékelése szenzorokkal, míg kvantum véletlenszámok esetén egy kvantum jelenségből erednek a számok. Ezeknek a számoknak az ábrázolásánál én a legelemibbnél döntöttem, amit a klasszikus számítógépeink ábrázolni tudnak, a biteknél. Így véletlenszámaimat valamekkora dimenzióval rendelkező bináris vektorként ábrázoltam és dolgoztam fel. Egészen pontosan két különböző méretű dimenzióval dolgoztam: 15 és 30. Ennek az oka az adatforrásaim korlátozott képessége.

3.2 Adatforrások és adathalmazok

A tesztjeim statisztikai jellegűek, így nagy mennyiségű adatot igényelnek. Mivel különböző típusú véletlenszám-generátorokat hasonlítok össze, ezért mindegyik típusra keresnem kellett egy megfelelő forrást. A forrásoknak viszont különböző hozzáférhetőségük, ingyen felhasználhatóságuk és képességeik voltak, ezért ezeknek a kiválasztása is egy hosszadalmas folyamat volt.

A véletlenszám-generátor típusok amiket összehasonlítottam:

1. Pszeudo véletlenszám-generátor
2. Igazi véletlenszám-generátor
3. Kvantum véletlenszám-generátor (QRNG)

A következőekben szeretném röviden összefoglalni az adatgyűjtés során szerzett tapasztalataimat. Minden adatforrás leírása után megtalálhatóak az onnan szerzett adathalmazom tulajdonságai. Háromféle tulajdonságot fogok említeni: az adott bit dimenzióban hányféle szám található, én hány számot szereztem az adatforrásból és, hogy az általam írt program által végzett egyediségvizsgálat után hány szám maradt meg az adathalmazból. Az utolsó tulajdonság igazából csak érdekesség és csak kis mértékben minősíti az adott generátort, mivel például a 15 bites esetben az összes lehetséges variációhoz képest nagy mennyiségű számot kértem le, így jelentősen megnőtt az esélye az ismétlődésnek, 30 biten viszont a lehetséges számoknak csak egy kicsi töredékét kértem el, és bár így sem jellemző egy generátorra az esetleges ismétlés, már sokkal többet jelent. Egy generált szám egyediségen végzett rendes statisztikai teszthez a

generálást és egyediség mérést még nagyon sokszor el kellene végezni, de ez is egy érdekes lehetőség lenne a véletlenszám-generátorok vizsgálatára.

3.2.1 PRNG Adatforrások

A legkönnyebb hozzáférést és testreszabást nyújtó PRNG-k a különböző programozási nyelvekbe beépített véletlenszám generátorok, de ezekből is többféle minőségű van. Az egyetemen több programozási nyelvet is tanítanak ezért sok választási lehetőségem volt. Végül a c++11 mellett döntöttem, mivel ez két fajta generátorhoz is hozzáférést nyújtott megfelelő egyszerűséggel, ezek a rand() függvény hívásával elérhető generátor és a jobb minőségű Mersenne Twister Engine, avagy mt19937. A rand() függvény által elért generátornál viszont már előbújtak a korlátok, amik a legtöbb másik generátornál is akadályozott, mégpedig a dimenzió korlát. A rand() függvény maximum 15 bites számokat képes generálni (16 amennyiben beleveszem a negatív tartományt). Ekkor utánanéztem, és láttam, hogy a legtöbb másik nyelvben is hasonló a dimenziókorlát. Egy másik, könnyebben kiküszöbölhető probléma a rand()-al az volt, hogy csak integer-eket generál. A Mersenne Twister Engine-el [11] nagyobb sikerrel jártam, ez ugyanis 32 bit-es véletlenszámokat is tud generálni.

3.2.1.1 Rand() függvény adathalmaza

A Rand()-al a fenti leíráshoz hűen csak 15 bites adatokat tudtam generálni, ennek az adathalmaznak a tulajdonságai:

- 15 biten elérhető variáció: 32 768
- Legenerált számok: 20 000
- Egyedi számok: 14 960

3.2.1.2 MT19937 adathalmazai

Mivel az MT19937 képes 30 bitet is generálni, ezért kértem egy 15 bites mintahalmazt is és egy 30 bites mintahalmazt is.

15 bites mintahalmaz tulajdonságai:

- 15 biten elérhető variáció: 32 768
- Legenerált számok: 20 000
- Egyedi számok: 14 900

30 bites mintahalmaz tulajdonságai:

- 30 biten elérhető variáció: 1 073 741 824

- Legenerált számok: 50 000
- Egyedi számok: 49 999

3.2.2 TRNG Adatforrás

TRNG adatforrásként a sokak által ismert Random.org [12]-ot használtam, amelyen keresztül elérhető a generátor, mely a honlap készítői szerint igazi véletlenszámokat generál atmoszférikus zajból, és melynek minősége jobb, mint a PRNG algoritmusoknak. A c++11-es PRNG-hez képest a hozzáférhetőség és a testreszabhatóság már jóval korlátozottabb volt a Random.Org esetében, ami nagymértékben azon is múlott, hogy megvolt az a lehetőségem, hogy a teljes verziót kipróbálhassam. Így a korlátok hasonlóak és még egy kicsit szigorúabbak is voltak, mint a PRNG-s esetemben. Mivel naponta csak 10 000 véletlenszámot lehet lekérni és maximum 32 biteseket, bár itt már lehetet bitsorozatként is generáltatni. Egy érdekes problémába is ütköztem a lekérés során, amikor is valami furcsa oknál fogva a generált számok megjelenítéséért felelős HTML-oldal változó mennyiségű extra 1-es bitet írt a véletlenszámaim elé, ami miatt egy külön formázó programot kellett írnom, mivel a másik megoldás a 10 000 számot kézzel szerkeszteni.

3.2.2.1 Random.Org adathalmaza:

A Random.Org is 32 bites számokat engedett generálni az ingyenes verzióval, így ezekből is két csoportot hoztam létre.

15 bites halmaz tulajdonságai:

- 15 biten elérhető variáció: 32 768
- Legenerált számok: 19 999
- Egyedi számok: 14 981

30 bites halmaz tulajdonságai:

- 30 biten elérhető variáció: 1 073 741 824
- Legenerált számok: 50 000
- Egyedi számok: 50 000

3.2.3 QRNG Adatforrás

A QRNG adatforráshoz hozzájutás volt a relatív legbonyolultabb, mivel ilyenhez nem igazán van online felületes hozzáférés, és mivel egy QRNG ritka és értékes eszköz, a használatuk legtöbbször fizetős. Szerencsére végül megtaláltam, amit kerestem az Australian National University (ANU) honlapján. Itt az egyetem saját QRNG-jéhez nyújtanak hozzáférést különböző módszerek által. Az én választásom egy pythonos interfészre esett, mivel ezt a nyelvet a neurális

hálózataim fejlesztésénél is használtam. Ennek az interfésznek a neve quantumrandom[13], Linux command line toolként is használható és az interneten keresztül éri el az ANU QRNG-ét. Maga az interfész könnyen kezelhető és egészen korlátlan, legalábbis nem találtam határt a generálható mennyiségnek, és a dimenzió határ is valahol 2^{2048} környékén húzódik. Az egyetlen probléma amibe belefutottam a kezelésnél a véletlenszerű biteknek a visszaadási formátuma volt, ami egy addig általam ismeretlen hexadecimális formátum volt, \xHH néven. Ennek a problémának a megoldása után a generátor könnyen kezelhetőnek bizonyult.

3.2.3.1 ANU QRNG adathalmazai

Itt háromfajta adathalmazt is generálhattam, hála a viszonylagos korlátlanságnak. Az egyik adathalmaz olyan hosszú sorokat tartalmaz, melyek a windowsos txt formátumnak nehezen kezelhetőnek bizonyultak.

15 bites halmaz tulajdonságai:

- 15 biten elérhető variáció: 32 768
- Legenerált számok: 20 001
- Egyedi számok: 14 974

30 bites halmaz tulajdonságai:

- 30 biten elérhető variáció: 1 073 741 824
- Legenerált számok: 50 000
- Egyedi számok: 50 000

Hosszú adathalmaz tulajdonságai:

- 2^{2048} variáció
- Legenerált: 100

3.3 Adatformázás

Az adatformázás folyamata több egyszerű, kisméretű program megírását jelentette, melyekre több okból is szükség volt: a Random.Org lekérési hibától az \xHH formátumig. A következőekben felsorolom ezeket a programokat, és egy rövid leírást adok hozzájuk.

1. c++11 véletlenszám generáló:
    a. Leírás: Ez a program generálja a PRNG adathalmazt, a generálás módszere állítható a Rand() függvény és MT19937-es megoldások között. Az adatmennyiség állítható, és a dimenzió is a korlátokon belül.

b. Bemenet: -

c. Kimenet: Egy txt file, ahol minden sor egy adott dimenzióval rendelkező bitvektor. A vektorok dimenziója a file során ugyanaz.

2. C++11-es kód egyediség vizsgálathoz:

a. Leírás: Egy brute-force egyediség vizsgáló. Megvalósítás röviden: Bemeneti számok beolvasása egy std::vectorba, majd amíg ez ki nem ürül átírás egy std::mapba, ahol a kulcs a bit vektor, az érték pedig az, hogy hányszor szerepelt.

b. Bemenet: Egy txt file, amiben soronként egy bitvektor szerepel. A file-on belül az összes bitvektornak ugyanolyan dimenziójúnak kell lennie.

c. Kimenetek: Két txt file: Az első tartalmazza azokat a vektorokat, amelyek egyediek, tehát amihez az std::mapban egyes érték tartozik. A második pedig tartalmazza azokat, amik ismétlődnek, mellettük az ismétlődés mennyiségével.

3. Java kód a Random.org-os számok formázására:

a. Leírás: A Random.org lekérésnél változó mennyiségű 1-est rak a generált bitvektor elejére. Ezeket metszi le a program.

b. Bemenet: Egy txt file, ami soronként egy formázatlan Random.org-os véletlen bitvektort tartalmaz.

c. Kimenet: Egy txt file, ami soronként egy formázott Random.org-os bitvektort tartalmaz. Itt a dimenzió a file-on belül konzisztens

4. Python kód az ANU QRNG eléréséhez:

a. Leírás: Ez a program változtatható paraméterű és mennyiségű véletlenszámokat kér le változtatható formátumban az ANU QRNG-jétől-

b. Bemenet: -

c. Kimenet: Egy txt file, ami a lekért paraméterű és mennyiségű véletlenszámot tartalmazza.

5. C++ kód a \xHH jelölés feloldására:

a. Leírás: Ez a program bitvektorba fejti az ANU QRNG által visszaadott biteket \xHH jelölésből bináris vektorba.

b. Bemenet: Txt file, mely soronként egy \xHH formátumu vektort tartalmaz.

c. Kimenet: Egy txt file, mely a kifejtett bináris vektorokat tartalmazza. Egyet egy sorban.

6. C++-os sorszámláló:

   a. Leírás: Megszámolja az adott file-ban a sorokat, amely az én esetemben az adatállomány méretét jelenti. Statisztikai szempontból fontos.

   b. Bemenet: Egy txt file.

   c. Kimenet: Standard outputra a sorok száma.

7. Python-os bitkép generáló egy képre Kohonen háló:

   a. Ez egy adott oszlop és sorszámú mátrixra működik, melynek elemei értékeiket az egész számok halmazából vehetik. Ezt alakítja át egy PNG-vé, ami 6 fajta színt használ. Feketét, fehéret, pirosat és a kék három árnyalatát. Jelentéseik le lesznek írva a hozzátartozó hálózat architektúrájánál.

   b. Bemenet: Egy txt file, ami egy mátrixot tartalmaz.

   c. Kimenet: Egy png, ami a mátrix képbe fejtését tartalmazza.

8. Python-os bitkép generáló mappára:

   a. Leírás: Ez a program a Konvolúciós Hálózat bemeneteit generálja. A felette lévő programtól két dologban különbözik: ,em csak egy filera működik hanem mappákra, és a bemeneti mátrix értékei csak 0 és 1 közötti értékek lehetnek.

   b. Bemenet: Txt file-ok, ami 0-ákból és 1-esekből álló mátrixot tartalmaznak.

   c. Kimenet: PNG file-ok, amit egy fekete fehér bittérképet tartalmaznak.

4 Az elvégzett fejlesztés bemutatása

Ebben a fejezetben fogom tárgyalni a munkám nagy részét. A modelljeimet, algoritmusaimat és architektúráimat, amiket részben kifejlesztettem, részben máshonnan felhasználtam. Mivel programoztam, ezért a teljesség megkívánja, hogy szót ejtsek az eszközökről melyeket használtam. Egy másik ok, amiért fontos kitérni a használt fejlesztőkörnyezetekre az az, hogy a nyelvválasztás is fontos lépése volt a munkámnak, amit itt a legegyszerűbb és a leghelyénvalóbb megmagyarázni még a fejezet elején.

4.1 Fejlesztő környezetek és technikai információk

Programozó munkám során kizárólag az alább felsorolt három fejlesztőkörnyezetet használtam eltérő mennyiségben.

4.1.1 Eclipse Java Oxygen

Az Eclipse [14]-t ebben a munkában nem igazán használtam, mivel csak egy Java programot írtam, azt is csak egy mellékfeladathoz, amely bár fontos volt, nem igényel túl sok figyelmet. A környezet nem használásának fő okát azonban a fejlesztési terület miatt mégis említenem kell. A neurális hálózatokban a szűk keresztmetszet gyakran a fejlesztési idő, amit bár Java-ban lehet optimalizálni, nem a legegyszerűbb. Ezért csak egy olyan feladatra használtam, ami nem igényel túl sok erőforrást.

4.1.2 Code::Blocks

Ebben a fejlesztőkörnyezetben [15] fejlesztettem a legtöbb mellékfeladatot végző szoftvert, valamint az architektúráimból a Kohonen hálót szinte teljesen, és a többrétegű perceptron (MLP)-nek egy kezdeti verzióját is. A c++ egy nagyszerűen és egyszerűen optimalizálható nyelv, ami a neurális hálózatok fejlesztése során igen jól jön. Ami nekem a legnagyobb hasznomra vált benne, az a memória kezelés feletti szinte tejhatalom, amely megmentette a Kohonen Hálózatos architektúrámat a biztos kudarctól, amikor is az a Tensorflow-os környezetben 6 GB memóriát megevett, míg c++-ban alig felett, sokkal nagyobb hálózatokra is.

4.1.3 Anaconda, Spider, Tensorflow

Az Anaconda[16] nem fejlesztő környezet, hanem egy platform fejlesztőkörnyezeteknek python nyelvhez. A telepítése sok drága időmet megette, már csak ezért is említést igényel. A választható fejlesztő környezetek közül én a Spider-t választottam, mert ennek tetszett meg a

legjobban a felhasználói felülete. A Tensorflow [17] pedig egy python nyelvkiegészítés külön neurális hálózatok építéséhez és kezeléséhez, ami sok figyelmet és időt igénylő munkától megkíméli a felhasználóit, ha azok értenek a működéséhez. A Tensorflow egy szokatlan futású nyelvkiegészítés még a python-hoz képest is, mely egy műveleti gráfot épít fel, majd futtat le. Előre megírt részeket tartalmaz sokféle neurális architektúrához, amit én fel is használtam.

4.1.4 Technikai információk

A tesztelés során a tesztelés gyorsaságára nagy befolyással volt, hogy éppen milyen hardveren futtattam a tesztet, a tesztelésnél külön jelöltem azt, hogy a rendelkezésemre álló két eszköz - egy asztali gép és egy laptop - közül melyiken futtattam. Az alábbiakban megadom az eszközeim fontosabb paramétereit:

Laptop:

- Processzor: Intel Core i5-5200U CPU @ 2.20GHz 2.19 GHz
- RAM: 4 GB
- Operációs rendszer: Windows 8.1

Asztali gép:

- Processzor: Intel Core i5-4440 CPU @ 3.10GHz 3.10 GHz
- RAM: 8 GB
- Operációs rendszer: Windows 8.1

4.2 Bevezető az architektúrákról

Az alapötlet teljesen váratlanul jött, viszont később egyre jobban megtetszett. Miért is ne lehetne neurális hálózatokkal tesztelni a véletlenszám-generátorokat? A neurális hálózatok nagyszerű függvény approximátorok és nagy általánosító képességgel rendelkeznek, így potenciálisan jó eszközei lehetnek egy véletlenszám minőségének vizsgálásában.

Az interneten való keresés a véletlenszámok teszteléséről neurális hálózatokkal határozottan eredménytelen volt. A legfigyelemfelkeltőbb egy fórumon feltett kérdés volt, ahol a kérdező felvetette az ötletet, az egyetlen válasz rá pedig az volt, hogy ne tegye. Ez több dolgot is jelentett a számomra. Elsőként azt, hogy szinte teljesen kitaposatlan területre tévedtem, tehát teljesen magamra vagyok utalva, azaz nekem kell előállnom architektúra ötletekkel, amelyek nagy valószínűséggel nem fognak eredményre vezetni, és nem sok segítségem lesz a kutatásban.

Ez viszont azt is jelentette, ha mégis sikerül valamilyen eredményt elérnem, az tényleg eredmény lesz, legalábbis számomra mindenképpen.

A következő fejezetekben leírom az architektúrák fejlesztését. Mindegyiknek a fejlesztése iterációkban történt, ahogy új és új dolgokra és hibákra jöttem rá az előző iterációval kapcsolatban. Egy iteráció dokumentálása az alábbiak szerint néz ki: az alapötlet, ami megindította vagy folytatta a folyamatot, a kutatómunka és elméleti háttér a lépés mögött. Az általam készített adaptáció, az architektúra tesztjei és ezeknek elemzése és végül, ahol szükségesnek érzem, a megvalósított kódrészletek.

A három architektúrám a következő:

1. Egy többrétegű perceptron hálózat, ami megpróbálja kikövetkeztetni egy bitsorozatban a következő bitet, és ennek a sikeressége alapján következtetést vonok le a generátor képességeiről.
2. Egy Kohonen hálózat, amely egy generátor által előállított számokat próbál klaszterezni, ennek a sikeressége alapján következtetés levonása a generátor képességeiről
3. Egy konvolúciós hálózat, amely bitképeket kap többféle generátor által előállított számokról, és mindegyik képről megpróbálja eldönteni, milyen generátortól jött.

A három architektúrámból kettő tanítása nagyon idő és erőforrás igényes, így nem mindig sikerült elérnem az architektúra potenciálját, de a határidőig befejezett tesztesetek tanúskodnak az architektúra és az ötlet érdemeiről.

4.3 Multi-Layer Perceptron (MLP)

4.3.1 Nulladik Verzió

Első körben még feleslegesen ambiciózus voltam, megpróbáltam az architektúrát megvalósítani c++ nyelven, a megvalósítás sikerült, viszont ekkor rájöttem, hogy a különféle optimalizációs eljárások megvalósítása nagyon nagy extra teher lenne, ami rámenne a TDK összértékére, és az egyetemi tanulmányaimra, nem megvalósításuk pedig a teszteseteim lehetséges variációjának terét csökkentené. Így inkább áttértem egy sokkal egyszerűbb módszerre, a tensorflow használatára, amivel az optimalizációs eljárások és sok más egyéb paraméter sokkal egyszerűbben variálhatók.

4.3.2 Első verzió, az architektúra születése.

4.3.2.1 Alapkoncepció

Az MLP hasznosságának az alapötlete volt az, ami az egész neurális hálózatokkal való tesztelést megindította, bár végül szerintem nem is ez az architektúra a legalkalmasabb a feladatra.

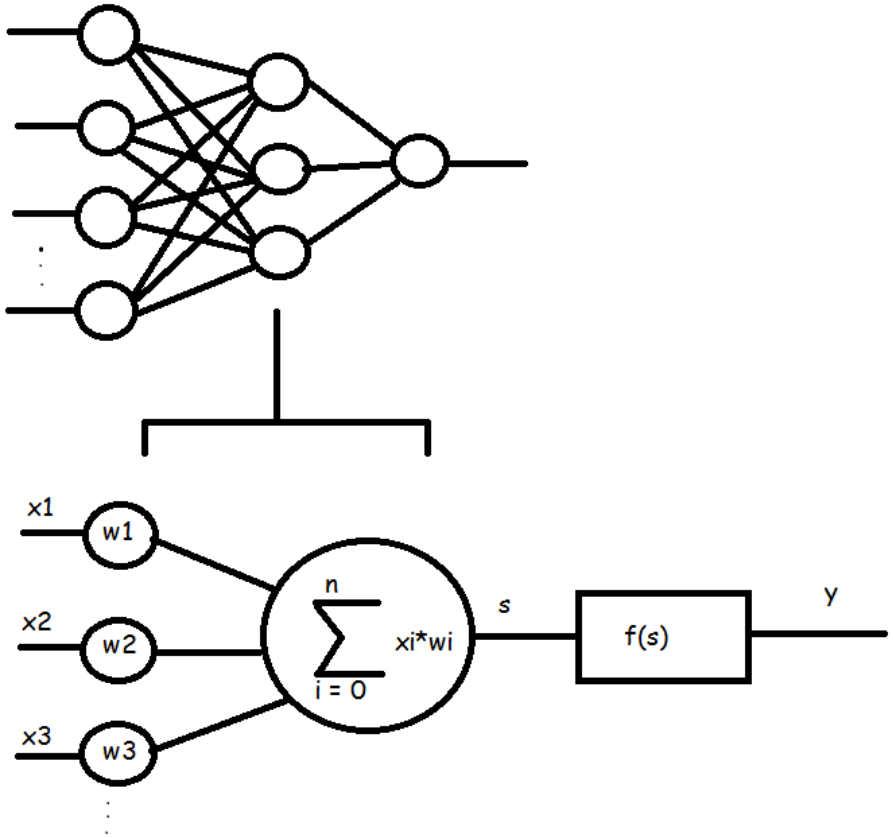
Az MLP-ket legtöbbször függvény vagy eloszlás approximációra használják és innen jött az ihlet, hogy hátha képes rátanulni egy véletlenszám-generátor mögötti eloszlásra.

4.3.2.2 Háttér és elmélet

Egy MLP tulajdonképpen a bemeneti vektorra vett súlyozott összeadók egymás mögé kötve, valamilyen aktivációs függvénnyel.

A bemenet esetemben egy véletlenszám-generátor utolsó n generált bitje, a kimenet pedig ezek súlyozott összegeire vett regresszió, ami egy 0 és 1 közötti lebegőpontos számot produkál, melynek jelentése az architektúrám tippje arra, hogy mi lesz a következő bit. Itt a 0 és 1 közötti szám valamelyik végértékhez vett közelsége még egy dolgot árul el, ez pedig a modellem biztossága a válaszában. Ez a véletlenszám-generátor elemzésben igen fontos, például az elemzésemben nem feltétlenül az lesz a rosszabb minőségű generátor aminek a számait a generátorom nagyobb arányban találta el. Fontos a biztossága a tippjében, ha például egy generátornál az esetek felében mindig eltalálta, hogy mi lesz a következő bit, viszont ezekben az esetekben a biztossága 0.49 és 0.51, az egy jobb minőségű generátor, mint amelyiknél az esetek egytizedében találta csak el, de akkor 0.05 és 0.95 bizossággal mindig eltalálta, hiszen az első eset csak azt jelzi, hogy szerencséje volt a tippeknél, míg a második esetben talált valami mintát, ami az esetek nagy részében fent is állt, így kiszámíthatóbbá téve a generátor által előállított

számokat.



1. ábra: Egy MLP és egy Perceptron (saját ábra)

Az architektúrát az 1. ábra segítségével mutatom be. Ennek az architektúrának az esetében nagy mértékű elméleti módosításra nem sok lehetőség van anélkül, hogy egy másik neurális architektúrába ne csúsznák, ezért az iterációk főleg a különféle paraméterkonfigurációkból fognak állni.

A paraméterek a következőek:

- L: Ez jelöli, hány rétege van az adott iterációban megjelenő MLP-nek. Értéke nagymértékben befolyásolja a háló általánosítási képességét. A hálózat mélységének is szokták nevezni, bele számolva a rejtett és kimeneti rétegek is.
- Ni : Ez jelöli, hogy az *i*-edik rétegben hány neuron található. Ha ez minden rétegben egyenlő, akkor egyszerűen *N*-el fogom jelölni. Ezt szokták a hálózat szélességének is nevezni.
- FC (Fully connected): Ennek értéke lehet igaz, hamis, drop out. Ha értéke igaz, az azt jelenti, hogy a hálózat *j*-edig rétegében minden neuronhoz kapcsolódik az *j-1*-edik réteg minden neuronja, ennek minden *j*-re igaznak kell lennie. Ha az

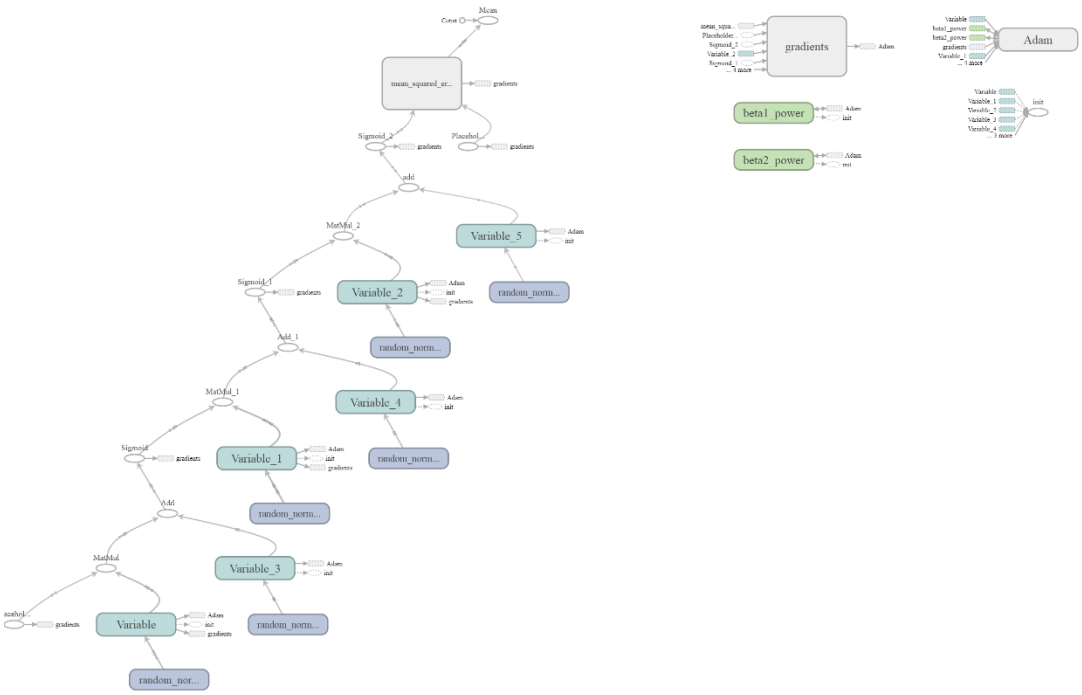
értéke hamis, akkor a fenti állítás nem igaz. Ha az értéke drop-out, akkor a futás közben néha egy-egy feleslegesnek értékelt neuront kiveszünk a hálózatból.

- f(x): Ezzel az aktivációs függvényt jelölöm, ami mindig regressziós függvény lesz mivel a megoldani kívánt probléma nem osztályozás. Szokásos értékei a szigmoid, logisztikus aktivációs függvények.
- TI (Training Iteration). Azt jelöli, hány iteráción keresztül tanítottam a hálózatot. Ennek a mennyiségétől függ nagyban a háló általánosító képessége, mert ha túl nagy, akkor rátanítottam a mintahalmazra, és nem lesz képes általánosítani, ha pedig túl kicsi, akkor nem tudja egyáltalán még, mit csinál.
- LR (Learning Rate). Azt jelöli, milyen gyorsasággal tanuljon a hálózat. A tanítási fázisban játszik szerepet csak. Ha túl nagy, akkor a hálózat túlugorhat a célfüggvényen néhány kiugró minta miatt, ha túl kicsi, akkor pedig soha nem éri el azt.
- CF (Cost Function). Ez azt jelöli, hogy mi a minimalizálni kívánt veszteség függvény, értéke szinte mindig $Mean^2$ lesz, azaz az átlagos hiba négyzet minimalizálása.
- Opt: Optimalizációs eljárás. Ez a tanítás mikéntjét jelzi. Tipikus értékei: Gradient Descent, Adam.

4.3.2.3 Implementáció

Az eredeti implementáció python nyelven tensorflow-al történt, egy github projekt [18] alapján.

A fent felsorolt paraméterek a kódból változtathatóak.



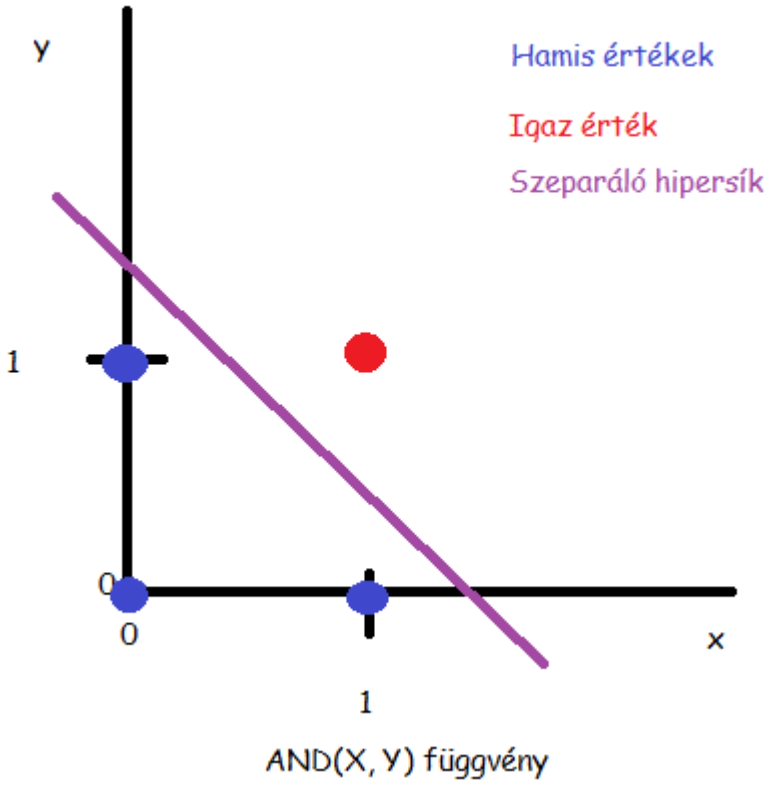
2. ábra: Három réteges MLP tensorboard[3] képe

A. 2- ábra a három réteges paraméterezést mutatja. Alulról felfele a rétegek a következőek: Első és második rejtett réteg, majd a kimeneti réteg. Végül pedig a veszteségi függvény operáció.

4.3.3 Tesztesetek és eredmények

A neurális hálózatokat leíró fejezetben említettem, hogy a hálózatoknak sokáig nagy akadályai voltak a lineárisan nem szeparálható problémák. Lineárisan nem szeparálható problémának nevezünk olyan osztályozó problémákat, melyekben a különböző osztályokba tartozó pontokat(adatokat) az adott dimenziójú térben nem lehet elkülöníteni egy hipersíkkal. Egy egyszerű példa lineárisan szeparálható problémára, egy 2 bites AND függvényre:

[3] A tensorboard a tensorflowhoz tartozó gráf vizualizáló eszköz.



3. ábra: Példa egy lineárisan szeparálható problémára

A tesztek iterációi miatt (egyszerű esetek, bonyolult esetek) újabb adathalmazokat kellet létrehoznom saját logikával, ami nagyon sok időt és főleg figyelmet igényelt. Egy teszt 20-60 vektort tartalmaz tanításnak, és 10-30 vektort tartalmaz validálásnak. A tanító minták 5 és 10 bitesek.

4.3.3.1 Egyszerű, determinisztikus példák

Teszt 1: AND logika

A háló a bemenetein 5 bites vektorokat kapott, a megtanulandó logika a következő: Ha a második és a negyedik bit 1 akkor a következő (hatodik) bit is 1 lesz. Paraméterek:

L	3
N	20
FC	Y
f(x)	Szigmoid
TI	2000

LR	0.001
CF	$Mean^2$
OPT	Adam

Teszt eredményei:

Prediction: [[0.18992972]] Answer: 1

Prediction: [[0.0001281]] Answer: 0

Prediction: [[0.99873513]] Answer: 1

Prediction: [[0.38789576]] Answer: 0

Prediction: [[0.9995365]] Answer: 1

Prediction: [[0.00970491]] Answer: 0

Prediction: [[0.99756867]] Answer: 1

Prediction: [[0.00020452]] Answer: 0

Prediction: [[0.00034146]] Answer: 1

Prediction: [[0.99597555]] Answer: 0

Az eredmények alapján egész jól néz ki a háló, csak kétszer rontott, tovább lehetne optimalizálni a tökéletes megoldásig, de nekünk most nem éri meg.

Teszt 2: XOR logika

Ez már egy lineárisan nem szeparálható probléma, így a megoldása a hálónak már nem lesz olyan egyszerű.

A második vagy negyedik bit 1 → kimenet 1.

Mindkettő 0 vagy 1 –y kimenet 0.

A paraméterek ugyanazok mint felül

Eredmény:

Prediction: [[0.8994225]] Answer: 0

Prediction: [[0.99972004]] Answer: 1

Prediction: [[0.0001939]] Answer: 0

Prediction: [[0.51073307]] Answer: 1

Prediction: [[0.00045813]] Answer: 0

Prediction: [[0.29327592]] Answer: 1

Prediction: [[0.09132411]] Answer: 0

Prediction: [[0.99978906]] Answer: 1

Prediction: [[0.99964345]] Answer: 0

Prediction: [[0.00694101]] Answer: 1

Itt már négy rossz válasz van ami a 10 esethez képest igazán rossz. Próbáljuk meg megváltoztatni a paramétereket!

Teszt 3: XOR logika 2

A feladat megegyezik a múltkorival.

Paraméter változások:

L	4

Prediction: [[0.7481131]] Answer: 0

Prediction: [[0.99996066]] Answer: 1

Prediction: [[0.00035405]] Answer: 0

Prediction: [[0.63282734]] Answer: 1

Prediction: [[0.00025857]] Answer: 0

Prediction: [[0.9976089]] Answer: 1

Prediction: [[0.00061797]] Answer: 0

Prediction: [[0.9999436]] Answer: 1

Prediction: [[0.99993896]] Answer: 0

Prediction: [[0.01394377]] Answer: 1

Itt máris jobban látszanak az eredmények, és már itt is csak két hiba van, most ezt is ennyiben hagyjuk bár a mintahalmaz bővítésével és a paraméterek finomításával itt is el lehet érni a tökéletes megoldást.

4.3.3.2 Bonyolult determinisztikus példák

Teszt 4: (x1 és x6) vagy (x3 és x7),

ahol xi az i-edik pozícióban álló bit értéke.

L	5
N	40
I	3000

Prediction: [[0.9999999]] Answer: 1

Prediction: [[0.99999833]] Answer: 1

Prediction: [[0.9999999]] Answer: 1

Prediction: [[0.999998]] Answer: 1

Prediction: [[7.1158763e-07]] Answer: 1

Prediction: [[0.14401166]] Answer: 1

Prediction: [[0.9999999]] Answer: 1

Prediction: [[0.9999399]] Answer: 1

Prediction: [[0.0074936]] Answer: 1

Prediction: [[0.9966497]] Answer: 1

Prediction: [[0.9995585]] Answer: 1

Prediction: [[0.9993869]] Answer: 1

Prediction: [[1.04292624e-07]] Answer: 0

Prediction: [[2.5926727e-06]] Answer: 0

Prediction: [[4.8403213e-05]] Answer: 0

Prediction: [[0.00245042]] Answer: 0

Prediction: [[0.21164708]] Answer: 0

Prediction: [[0.01339246]] Answer: 0

Prediction: [[0.05650783]] Answer: 0

Prediction: [[7.828739e-05]] Answer: 0

Prediction: [[0.99999976]] Answer: 0

Prediction: [[0.00595652]] Answer: 0

Prediction: [[0.73211914]] Answer: 0

Prediction: [[0.999998]] Answer: 0

Itt 27 mintából 6 hiba és 1 bizonytalan eredményt kaptunk, ami jelenleg elég jó, a későbbiekben még optimalizálunk.

Teszt 5: (x1 és x6) xor (x3 és x7),

Prediction: [[0.7912386]] Answer: 0

Prediction: [[3.1307346e-07]] Answer: 0

Prediction: [[0.02644271]] Answer: 0

Prediction: [[5.561807e-08]] Answer: 0

Prediction: [[1.6462263e-05]] Answer: 1

Prediction: [[1.5739453e-05]] Answer: 1

Prediction: [[0.9999981]] Answer: 1

Prediction: [[0.9988207]] Answer: 1

Prediction: [[0.8885434]] Answer: 1

Prediction: [[0.9999515]] Answer: 1

Prediction: [[5.8618247e-07]] Answer: 1

Prediction: [[2.7399114e-05]] Answer: 1

Prediction: [[6.8953995e-08]] Answer: 0

Prediction: [[1.079266e-06]] Answer: 0

Prediction: [[0.4964758]] Answer: 0

Prediction: [[0.09641613]] Answer: 0

Prediction: [[0.99372333]] Answer: 0

Prediction: [[7.432992e-07]] Answer: 0

Prediction: [[0.0467871]] Answer: 0

Prediction: [[0.00075981]] Answer: 0

Prediction: [[0.8837537]] Answer: 0

Prediction: [[5.914919e-07]] Answer: 0

Prediction: [[0.99535453]] Answer: 0

Prediction: [[0.999956]] Answer: 0

9 hibás a 25-ből, bár nem a legjobb eredmény de jelenleg még megfelem mert nem ez az adathalmaz a teszt célja.

4.3.3.3 PRNG

MT19937 30 bites adathalmaz.

L	6
N	50
I	500

A nagy adazhalmaz és a kevés rendelkezésre álló idő miatt kevés iterációra maradt idő de az eredmények így is magukért beszélnek.

Prediction: [[0.60260653]] Answer: 0

Prediction: [[0.2904435]] Answer: 0

Prediction: [[0.64400697]] Answer: 1

Prediction: [[0.8186153]] Answer: 1

Prediction: [[0.52808374]] Answer: 1

Prediction: [[0.5846527]] Answer: 0

Prediction: [[0.41798985]] Answer: 0

Prediction: [[0.590968]] Answer: 1

Prediction: [[0.86552393]] Answer: 0

Prediction: [[0.68960786]] Answer: 1

Prediction: [[0.5667239]] Answer: 1

Prediction: [[0.43813363]] Answer: 1

Prediction: [[0.2270478]] Answer: 1

Prediction: [[0.55739254]] Answer: 0

Prediction: [[0.89820415]] Answer: 1

Prediction: [[0.81189644]] Answer: 0

Prediction: [[0.81512153]] Answer: 0

Prediction: [[0.22497037]] Answer: 0

Prediction: [[0.31669167]] Answer: 1

Prediction: [[0.28733683]] Answer: 1

Bár míg a nagy részét nem találta el, és egész bizonytalan volt a tippjében, néhány nagyon magabiztos válasza igaz lett, azaz itt ki tudott nyerni valami mintát a mögöttes eloszlásról. További paraméter finomítással és több tanítási iterációval, a találati arány minden valószínűséggel jobban közelítené meg az egyet. Ami még jobban megerősíti az architektúra sikerességét az a következő információ: A számok egyediek, a validációs halmaz vektorai nem részei a tanító mintahalmaznak.

4.3.3.4 TRNG és QRNG

A TDK határideje előtti utolsó éjszaka ennek a két tesztnek az eredményei sajnos egy másolási hiba áldozatául estek, így nem tudom bemutatni őket. A teszt eredményei rosszabbak voltak mint a PRNG-é, a háló sokkal bizonytalanabb volt mindegyik válaszában, bár néhányat így is eltalált elég nagy magabiztossággal.

4.4 Kohonen hálózat/Self Organising Map (SOM)

Ez az a hálózat, aminek a megvalósításával a legtöbb időt töltöttem, ami a legnagyobb nehézségekkel járt, amihez a legtöbb újítást tettem, mind kódban mind elméletben, és amire végérvényben a legbüszkébb is vagyok.

4.4.1 Nulladik verzió

Ezen architektúrám nulladik verziója az MLP nulladik verziójának a fordítottja, első próbálkozásom során egy tensorflow tutorial [19] alapján próbálkoztam megvalósítani egy Kohonen hálót, vagy SOM-ot. A tutorial követése során minden rendben ment, és bár egy kis idő eltelt, de végül az internetes forrás példájára le is fordult és jó eredményt is kaptam. Aztán megpróbálkoztam a c++ által generált 14 000 körüli számossággal rendelkező adathalmazommal. A példa háló reakciója erre az volt, hogy két óra után elkezdte elhasználni a laptopom RAM kapacítását majd négy óra végeztével 2.5 GB-nál elhasználta az összeset. Ekkor az egész architektúrát úgy, ahogy volt átmásoltam az asztali számítógépemre, ami több RAM-al rendelkezik és újra lefuttattam. Egy bő másfélóra futás után itt is elkezdte megenni a felhasználható memóriát majd még négy órával később elhasznált 5.9 GB-ot és még mindig kellett volna neki. Ekkor döntöttem úgy, hogy megvalósítok egy sajátot egy minta alapján c++-

ban és egy bő két nap elveszítése után Code::Blocks-ban próbálkoztam újra egy lépésről lépésre tanító weboldal segítségével [20].

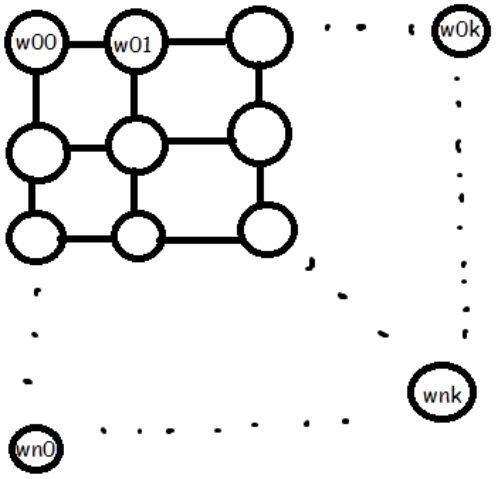
4.4.2 Első verzió, klasszikus Kohonen kálózat euklidészi távolsággal

4.4.2.1 Alapkoncepció

Ez az ötlet jutott eszembe utoljára, és erre is voltam akkor a legbüszkébb. A Neurális hálózatok című tantárgy jegyzetének [4] olvasásánál, a felügyelet nélküli tanulásnál láttam egy Kohonen hálózatot, ami képes a klaszterizálásra. Ekkor merült fel bennem az ötlet, hogy egy eloszlást szintén jellemez az általa gyakran felvett hasonló értékek, és, hogy amennyiben egy véletlenszám-generátor tendenciát mutat arra, hogy egymáshoz hasonló értékeket generáljon, az egy potenciális gyengesége, támadó felülete a generátornak, és ezáltal a generátort alkalmazó biztonsági rendszernek is.

Mit is számítok hasonlónak? Amikor ugyanazokban a pozíciókban ugyanolyan értékek szerepelnek.

4.4.2.2 Háttér és elmélet



4. ábra: Egy Kohonen háló topológiája (saját ábra)

A Kohonen háló topográfiáját 2. ábrán szemléltetem. Egy Kohonen hálózat sokban különbözik egy tipikus neurális hálózattól. Először is nem jellemző a többrétegűség. A topológia általában (és az én esetemben is) egy kétdimenziós rács rácspontjaira helyezett neuronokból állnak, ahol egy neuron csak egy súlyvektor.

A tanítás célja a SOM-oknál egy többdimenziós adathalmaz vizualizálása és klaszterizálása, mégpedig úgy, hogy az egymáshoz hasonló adatok egymáshoz közel essenek a rácson. A hasonlóság egy függvénnyel megfogalmazható. A tanítás versengéssel kezdődik, ahol

a bementi mintahalmaz egy adott elemére meghatározzuk a Best Matching Unit-ot (BMU), ami a mintához legjobban hasonló súlyvektorú neuron, majd a BMU környezetét, magát is beleértve módosítjuk úgy, hogy hasonlítson ehhez a vektorhoz. Majd ezt iteráljuk, és a rácsháló pontjain elhelyezkedő súlyok egy olyan felületet fognak kiadni, ahol a hasonlósági függvény által hasonlónak ítélt egymáshoz közel fognak elhelyezkedni, míg az egymástól eltérőek távolabb, a hozzájuk hasonlóak közelében.

4.4.2.3 Implementáció

Az implementációm alapja egy már létező kohonen háló kódja volt, ami később több szempontból is alkalmatlannak bizonyult, így a későbbi verzióim már nem is hasonlítanak az eredetire, az elnevezések viszont megmaradtak.

Implementációban fontos fogalmak:

- Szomszédsági sugár: Egy iterációnként csökkenő változó, ami egy neuron szomszédságának nagyságát jellemzi.
- Szomszédság: Egy neuronnak a rácson közeli szomszédai, amik a szomszédsági sugárnál közelebb vannak.
- Időkonstans: Ez a tanítási iterációk és a teljes rács átmérőjének a függvénye, fontos a szomszédsági sugár kiszámításában
- Befolyás: Egy neuron egy másik neurontól való távolságát jelzi a szomszédságon belül, amely a súlyok módosításánál játszik szerepet.
- Tanulási ráta: Egy változó, ami időben csökken, minden mástól független és szinten egy súly módosításának mértékét befolyásolja.

Ebben a verzióban az implementációm a következő volt:

A rácsháló neuronokat tartalmaz, amelyek tudják magukról a rácshálón belüli koordinátáikat, és képesek kiszámolni egy adott vektorhoz vett euklideszi távolságukat:

$$\sqrt{\sum_{i=0}^{n}(input_i - w_i)^2},$$

ahol *n* a bemeneti vektorhalmaz dimenziójának a száma, input egy bemeneti vektor és w az adott neuron súlyvektora. Ezen kívül egy megadott célvektorhoz képesek igazítani a súlyvektoraikat, a tanulási ráta és a befolyás függvényében.

A háló inicializálásakor sorsolunk véletlenszámokat 0 és 1 között a háló minden neuronjának vektorként

A tanítás egy iteráció pedig a következőképpen néz ki:

A bemeneti halmazból sorsolunk egy vektort, ehhez a bemeneti vektorhoz megkeressük a legkisebb euklideszi távolsággal rendelkező neuront, ez lesz a BMU. Majd megnézzük, most éppen mennyi a szomszédsági sugár, és a szomszédsági körébe eső neuronoknak módosítjuk a vektorait úgy, hogy az input vektorhoz hasonlítsanak, viszont minél távolabb van egy neuron a BMU-tól, annál kisebb mértékben mozdítsuk el a neuron súlyait az input vektor felé. Végül csökkentjük a szomszédsági sugarat és a tanulási rátát és kezdjük a következő iterációt addig, amíg véget nem ér a tanítás. Ezután minden bemeneti vektorra megkeressük a BMU-t és kiírjuk a rácsok egy txt-be az alábbi módon: ha egy neuronra esik bemeneti adat, akkor 1-et írunk az adott pozícióba, ha nem, akkor 0-át.

4.4.2.4 Programszerkezet és kód

Röviden a c++-ban írt kódról.

Fileszerkezet:

- Headerek
    - Cnode: Ez tartalmazza egy neuron implementációt a hálón belül, főbb függvényei a távolságszámítás és a súlymódosítás
    - Csom: Ez valósítja meg a rácshálót, főbb függvényei az Epoch és a FindBestMatchingNode.
    - Ccontroller: Ez az osztály irányítja a Csom működését, ő hívja meg az Epoch függvényt, tölti be az adatot és írja ki a végeredményt.
    - constants: Ebben a headerben vannak deklarálva a program legfontosabb konstansai mint a gridWidth, gridHeight, constSizeOfInputVector, constStartLearningRate, constNumIteration
    - utils: eredeti tutorialból megmaradt header, két függvényét használom még , a RandInt() és a RandFloat() függvényt ami leegyszerűsíti a véletlenszám generálásokat
- Source-ok, fenti headereket megvalósító fileok.
    - Ccontroller, Csom, main

A kód nagy részét megmutatni fölösleges mert egyszerű, de a követhetőség érdekében itt van egy tanítási ciklus kódja. A kód egy részéhez kommenteket írok, de nagyrészt önmagyarázó.

```
//preiter: A második fázis iterációinak száma
```

```
bool Csom::SecondPhaseEpoch(const vector<vector<int> > &data, int preiter)
{
  //Időkonstans ami a szomsédsági sugár méretének a változásának a
mértékét
  //befolyásolja
  double timeconst = preiter/log(MapRadius);
  double learnrate = constStartLearningRate;


  //Megnézzük, hogy megfelelő méretű-e a bemeneti vektorok tere
  if (data[0].size() != constSizeOfInputVector) return false;


  //tanítási ciklus
  for(int iter = 0; iter < preiter; iter++)
  {
    if( iter % 1000 == 0) cout << "\n2: " << iter;

    //Sorsolunk egy vektort, amin tanítunk
    int ThisVector = RandInt(0, data.size()-1);

    CNode* cn = FindBestMatchingNode(data[ThisVector]);

    //Kiszámítjuk a szomszédsági sugarat
    double neighradius = (double)MapRadius *
exp((double)(iter)/(double)timeconst);

    //Kiszámoljuk a neuron környezetére a befolyást és beállítjuk a
    //súlyokat.
    for (int n=0; n<m_SOM.size(); ++n)
    {
      double DistToNodeSq = (cn->X()-m_SOM[n].X()) *
                            (cn->X()-m_SOM[n].X()) +
                            (cn->Y()-m_SOM[n].Y()) *
                            (cn->Y()-m_SOM[n].Y()) ;

      double WidthSq = neighradius * neighradius;

      //Ha a csomópont a távolsága alapján a szomszédságba esik
      if (DistToNodeSq < (neighradius * neighradius))
      {

        //kiszámítjuk a befolyását
        double infl = exp(-(DistToNodeSq) / (2*WidthSq));

        m_SOM[n].AdjustWeights(data[ThisVector],
                               learnrate,
                               infl);
      }

    }


    //csökkentjük a tanulási rátát
    learnrate = constStartLearningRate * exp(-(double)iter/(preiter -
iter));
  }
  //Sikerült-e tanítani
  return true;
```

```
}
```

4.4.2.5 Tesztek, eredmények és értékelés

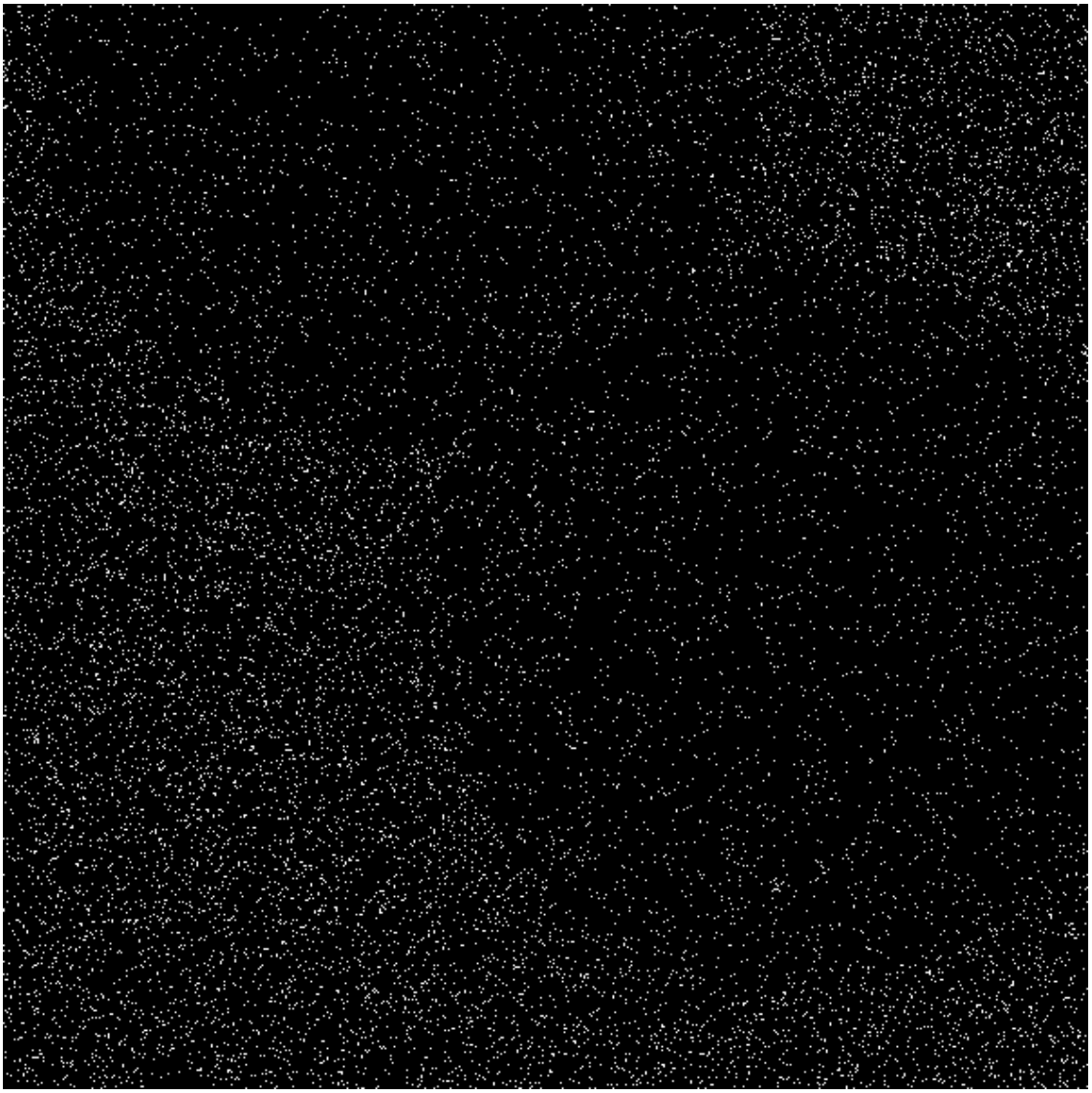
A fejezetben szereplő ábrák értelmezése: ahol fehér pixel van, ott a hálózaton adatvektor esik, ahol fekete, oda nem.

Ezt az architektúrát három adathalmazon teszteltem, a c++ által generált 15-bites halmazon, az ANU 30-bites halmazán és végül egy saját, 12 darab 10 bites vektort tartalmazó halmazon.

Az eredmények:

1. **teszt**: c++15bit adathalmaz, 2 iteráció:

Futásidő asztali gépen: ~10 perc

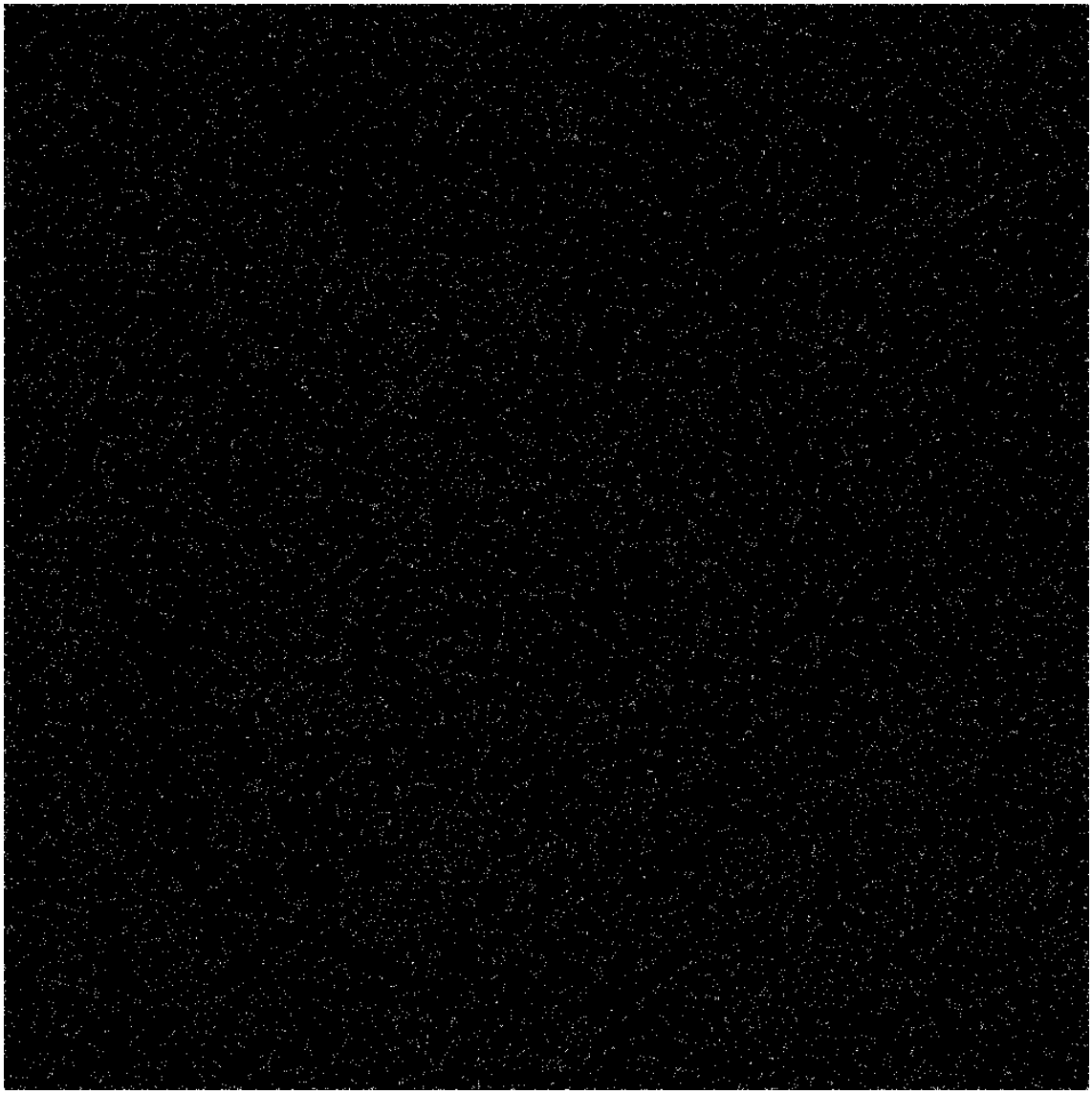


5. ábra : Kohonen háló első verziójának eredménye két iteráció után a 15 bites c++ adathalmazon

Értékelés: a hálózat úgy viselkedik, ahogy elvártuk, két BMU-t választottunk ki, és arrafelé igazítja a vektorokat, a képen szépen látszik a két szomszédság ahol igazítottunk. Próbáljuk ki nagyobb iteráció számmal.

2. **teszt** c++15bit adathalmaz 20 000 iteráció

Futásidő asztali gépen: ~4.5 óra

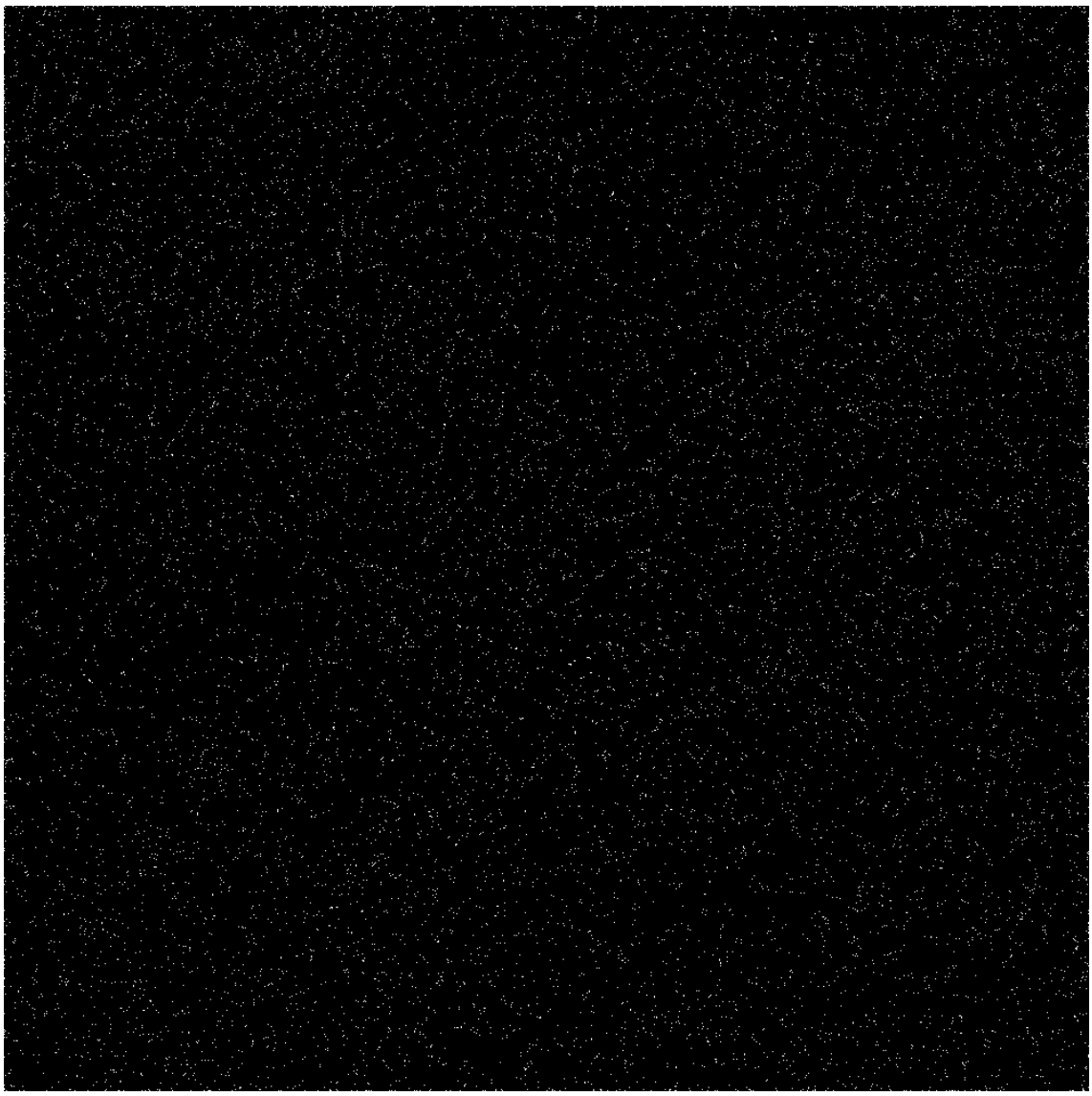


6. ábra: Kohonen háló első verziójának eredménye 20 000 iteráció után a 15 bites c++ adathalmazon

Értékelés: Nem látszódnak klaszterek egyáltalán. Akkori véleményem: Lehet, hogy mivel csak 20 000 iterációm van és ~17 000 vektorom és mivel véletlen szerűen sorsolok vektort minden iterációban, ezért nem kerül sor egy csomóra. De az is lehet, végül is vannak benne klaszterek csak nem látszódik, mivel vannak benne feketébb részek, nézzük meg a kvantum véletlenszámokra.

3. **teszt** ANU15bit adathalmaz, 20 000 iteráció

Futási idő asztali gépen: ~4.5 óra

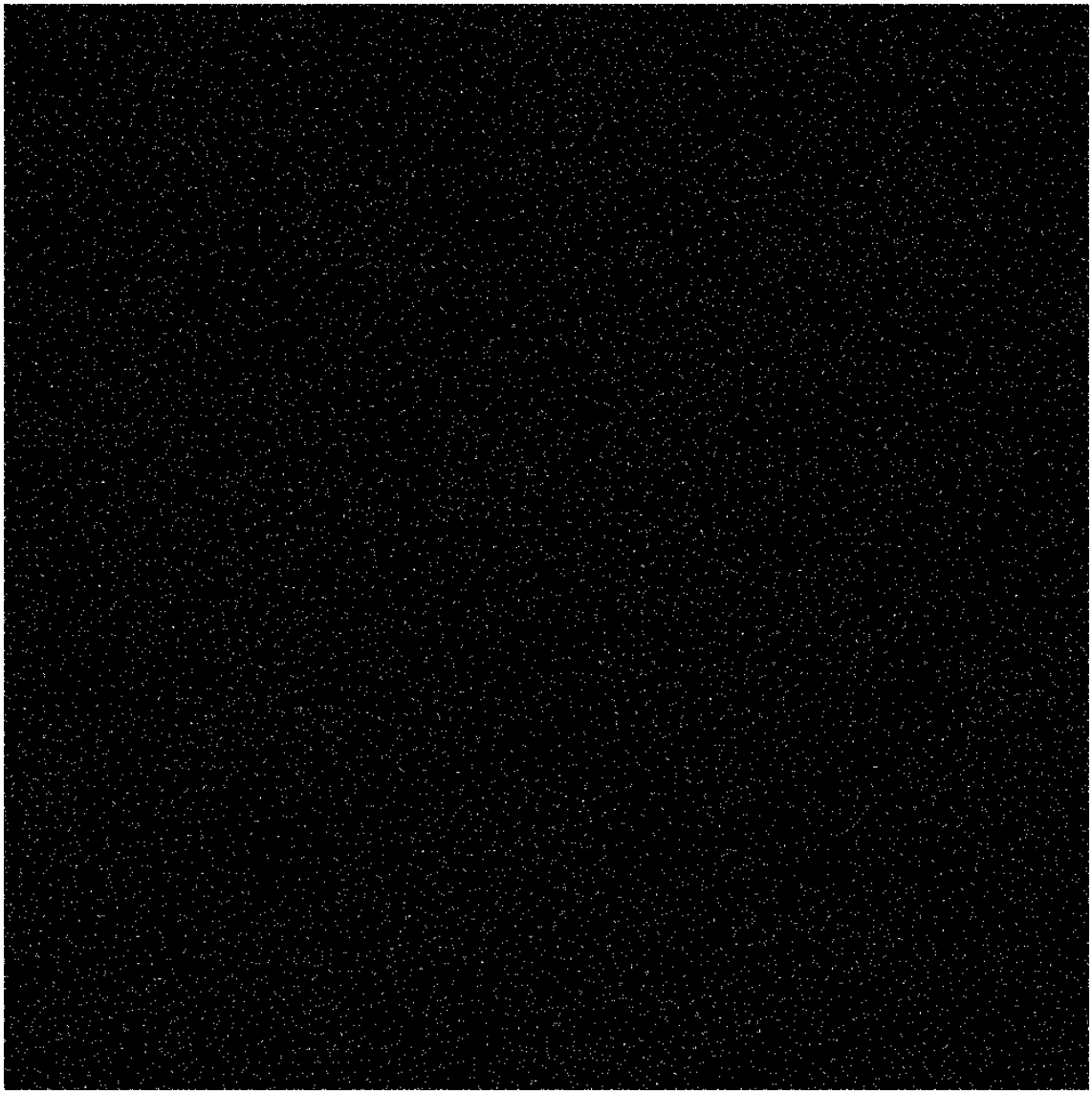


7. ábra: Konohen háló első verziójának az eredménye az ANU 15 bites adathalmazon 20 000 iteráció után

Értékelés: Itt sem látszik semmi, biztosan tényleg az iterációkkal lesz baj. Több iteráció!

4. **teszt** c++15bit adathalmaz 300 000 iteráció

Futásidő asztali gépen: ~1,75 nap



8. ábra: A Kohonen háló első verziójának az eredménye a c++ 15 bites adathalmazon 300 000 iteráció után

Értékelés: Semmi sem utal a képen a javulásra, sőt, az egyenletes eloszlástól sem várna az ember ilyen képeket. Nézzük meg egy kicsi adathalmazra, amit tudok ellenőrizni.

5. **teszt**, saját adathalmaz, 1000 iteráció

Futási idő asztali gépen: ~15 perc

9. ábra : Kohonen háló első verziójának eredménye 1000 iteráció után a saját adathalmazon

Saját adathalmazról tudnivalók: Két klasztert tartalmaz, melyek egymástól max 2 Hamming távolságra vannak.

Értékelés: Valami baj van és az előző két napot feleslegesen elpocsékoltam, mert nem néztem meg kicsi esetekre a hálómat.

4.4.3 Második verzió, klasszikus Kohonen hálózat Hamming távolsággal

Az előző verzió végén rájöttem, hogy van egy potenciálisan nagy hibám. Az euklideszi távolság nem ugyan az, mint a Hamming távolság, és nincs is szabályos aránylás a kettő között az esetemben, mivel a bemeneti vektoraim bináris vektorok, míg a súlyaim lebegőpontos vektorok. Így viszont a Hamming távolságnak nincs értelme, mert az esetek nagy részében maximális lenne egy sima összehasonlításnál.

4.4.3.1 Alapötlet

Mi lenne, hogyha a súlyaim bináris vektorok lennének. A felvetés szép volt, mert akkor a Hamming távolságnak lenne értelme a kettő között és azt is kapnám, amit szeretnék. Ezt gyorsan le is implementáltam és már futtattam is mielőtt eszembe jutott, ez miért rossz megközelítés. Így elvesztettem a neurális hálózatok fokozatos tanulási képességét, mert a ciklusban a bitek 0→1 és 1→0 átmenetet hajtottak végre egy lépés alatt, ami a hálót teljesen tönkrevágta.

Így a következő lépés az maradt, hogy csak a távolság számításnál kezelem őket bináris vektorként, tehát amikor a Hamming távolságot veszem, akkor egyszerűen ha a súly kisebb mint 0.5 akkor 0, ha meg nagyobb mint 0.5 akkor 1. Ezzel az a baj, hogy nagyon radikális a változás, azaz ha egy vektor 0.49, akkor egy kicsi mozdulás is átlendítheti a másik kategóriába, és egyébként is nehéz azt mondani 0.50000001-re, hogy 1. Az implementáció teljesen eredménytelen volt.

Így következő lépésként határokat szabtam a kerekítésnek: mettől számítson egynek, mettől nullának és mettől egyiknek sem.

4.4.3.2 Elmélet

Maga az architektúra ebben a lépésben nem sokat változott, lényegi eltérés az volt, hogy az Euklideszi távolság helyett Hamming távolságot használtam.

4.4.3.3 Implementáció

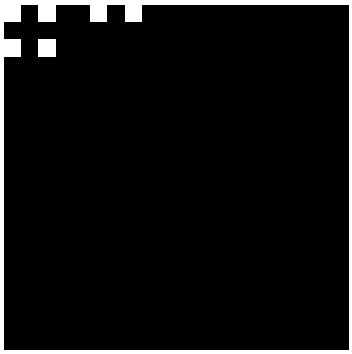
A Hamming távolság kiszámítása és a határok:

```
double distance = 0;
for (int i=0; i<n_Weight.size(); ++i){
        int WeightToBit = n_Weight[i] < 0.05 ? 0 : n_Weight[i] > 0.95  ? 1 : 2;
        distance += WeightToBit == InputVector[i] ? 0 : 1;
}
```

4.4.3.4 Teszt, eredmény, értékelés

Remélve, hogy az új ötletemmel már működni fog a háló, megint lefuttattam a sajátadathalmazomon a tesztet még egyszer.

Eredmény:



10. ábra : A Kohonen háló második verziójának eredménye a saját adathalmazomon.

Mintha már alakulna valami, de mégsem, ráadásul egy csomó adatot egymásra pakolt, amit viszont nem jelzek sehogy, ezért olyan, mintha nem is rakta volna fel őket. Megint visszakerültem a start mezőre és gondolkodhattam előről.

4.4.4 Harmadik verzió, BinBatch

4.4.4.1 Alapötlet

Kezdtem azt gondolni, hogy az architektúra nem alkalmas bináris vektorok klaszterizálására, így az interneten néztem utána, mások hogy csinálják. Rengeteg ne próbáld válasz után találtam egy cikket [21], ami referenciát adott egy másik cikkre [22] a 2000-es ESANN konferencián, ami Kohonen hálót alkalmaz bináris adatokra.

4.4.4.2 Háttér és elmélet

A cikk alapötlete a következő pár lépés volt, lerövidítve és egyszerűsítve:

Használjuk a Hamming távolságot a BMU megtalálására (amit kiegészítettem a határhúzással)

1. Számoljuk ki minden bemeneti vektorra a BMU-t, és tartsuk számon, melyik rácsponthoz vannak rendelve
2. Számoljuk ki egy vektorra a szomszédságának a nagyságát.
3. Az adott vektor összes szomszédjára és magára képzett bemeneti vektort fogjuk össze egy listába és számítsuk ki erre a medián központot [4]és erre felé mozgassuk a súlyait.

Ezt a három lépést ismételjük meg minden csomópontra, majd a fenti lépést hajtsuk végre addig, amíg a rendszer nem stabilizálódik.

4.4.4.3 Implementáció

Ennek az elméletnek az implementációjához sok mindent meg kellett változtatnom a modellemben és a programomban. Egy új, fontos függvény lépett be mint a MedianCentre() ami a medián központot számítja ki egy adott halmazra, valamint az Epoch() függvény teljesen megváltozott, mivel itt eddig csak egy kisorolt adatvektorra kelett megnézni a BMU-t, majd csak ezzel a BMU-val és környezetével kellett foglalkozni, ezután pedig minden adathalmazzal és neuronnal kellet foglalkozni egy lépésen belül.

---

[4] A programomban a súlyozás nélküli medián központot használtam, ami azt a vektort jelenti, amitől az adott vektorhalmaz elemeinek a távolságának az összege minimális. Súlyozás nélkül ezt könnyű meghatározni, mert minden eleme a medián központnak az a szimbólum, amit az adott pozícióban a vektorhalmaz vektorai többször választottak.

Valamint az eredmény lementésének és kijelzésének a módját is meg kellett változtatnom, hogy az egymásra csúszott adatok számát is tudjam jelezni, ezt ezután színnel kódoltam.

```
vector<int> Csom::MedianCentre(vector<vector<int> > &data)
{
    vector<int> ret;

    for(int i = 0; i < constSizeOfInputVector; i++)
    {
        int num0 = 0;
        int num1 = 0;
        for(int j = 0; j < data.size(); j++)
        {
            data[j][i] == 0 ? num0++ : num1++;
        }

        num0 > num1 ? ret.push_back(0) : ret.push_back(1);
    }

    return ret;
}
```

4.4.4.4 Tesztek és eredmények.

Ezt a megoldást szintén a saját halmazomra futattam le, viszont még mindig nem akart működni.

4.4.5 Végső, saját verzió, három tanítási fázisú Kohonen hálózat bináris adatok klaszterizálására.

4.4.5.1 Alapötlet

Végre rájöttem, mi volt végig az egyik baj a klaszterizálásommal, ami hozzájárult ahhoz, hogy nem működött egyik verzióm sem. A probléma abban rejlik, hogy nem végeztem előfeldolgozást a hálómon. Nézzük meg az eddigi verziókat[5]!

A második verzióban kiválasztunk egy adatot, egy kicsit módosítunk a BMU-ja környezetén, hogy hasonlítson rá, majd kiválasztunk egy következőt és ismételjük. Itt az a probléma, hogy véletlen szerűen generált súlyokkal indulunk, amit az elején messze vannak a 0 és 1 szélsőséges értékektől, ezért amikor lerakjuk a következő adatot, van rá esély, hogy bár nem hasonlít az elsőre, mégis mellé kerül, így a BMU és környezet módosításával belezavar az első adat klaszterébe.

[5] Az első verziót kihagyom, mert ott bezavar az euklideszi távolság.

A harmadik verzióban pedig még nagyobb probléma van, lerakja az összes vektort BMU-kra, amik akárhogy helyezkedhetnek el egymást képest az elején, aztán ezeknek veszi a medián közepét és erre fele tolja őket, így tényleg kialakulnak klaszterek, de ezen tagjai között nincs olyan hasonlóság, amire nekem szükségem van.

Ekkor jött az ötlet, hogy meg kellene keresnem az adathalmazomban a néhány legkülönbözőbb adatot, ezeknek kialakítani a környezetüket és azután rájuk helyezni a többit.

4.4.5.2 Elmélet

A fenti okokból kifolyólag tehát kidolgoztam egy heurisztikát, ami viszonylag gyorsan kiválaszt az adathalmazból egy olyan részhalmazt, amik viszonylag távol vannak egymástól.

A heurisztika a következőképpen néz ki: Sorsoljunk ki egy vektort, keressünk meg a tőle viszonylag nagy távolságra lévő vektorokat, majd ugyanezt tegyük meg a többire is. Fogjuk ezeket a vektorokat (mindegyik egyedi), és szűrjük ki belőlük azokat, amik egymástól nincsenek nagy távolságra.

Így megkapunk az adathalmazunkból egy olyan részhalmazt, amelyek viszonylag nagy távolságra vannak egymástól. Ezeket mutassuk meg a hálónknak néhányszor.

Ekkor kialakulnak kezdetleges klaszterek.

Ezután mutassuk meg az összes adatvektort a hálónak a második verzió alapján, ekkor kialakulnak a klaszterek az új vektorokkal, és néhány újabb klaszter is esetleg.

Végül futtassuk le a harmadik verziót néhányszor, hogy a súlyvektorok medián központ felé való eltolása által egyenletesebb legyen a felület.

4.4.5.3 Implementáció

Ennek a verziónak az implementálása meglehetősen egyszerű volt, hiszen az eddigi verzióknak az implementációit kellett felhasználnom csak benne.

A némi újdonságot a heurisztika megvalósítása, és az egyenkénti tanítás jelentette. Ami lényeges változás volt az az, hogy a konstansok között megjelent három iteráció is, és a Csom osztálynak három epoch függvénye lett, ami a három fázist jelentette.

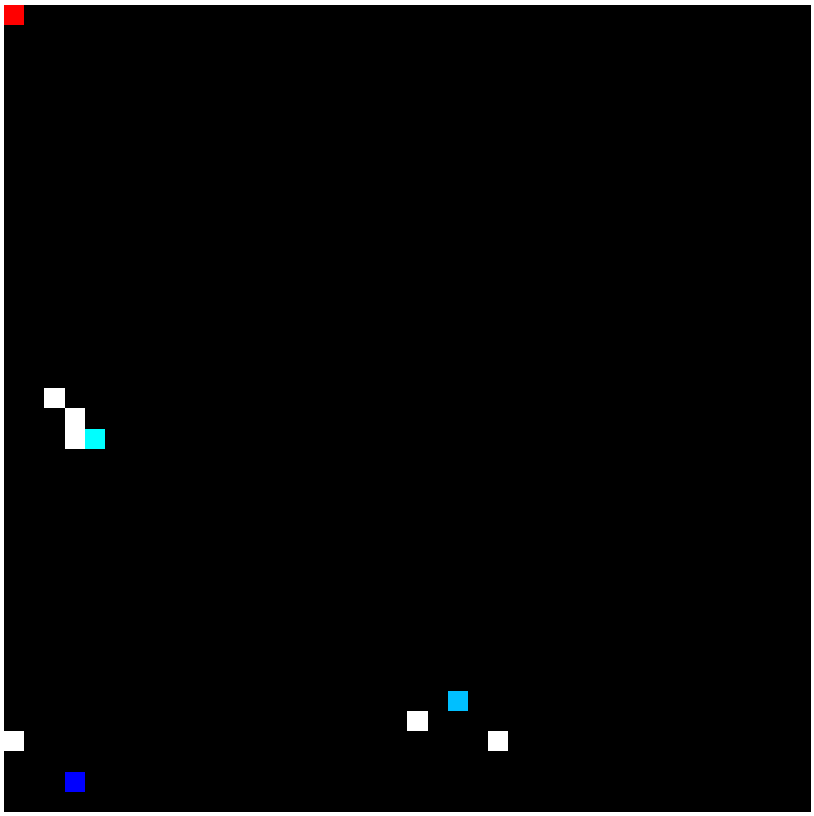
A heurisztika függvény:

```
int ThisVector = 0;
if(m_TrainingSet.size() > 1)
   ThisVector = RandInt(0, m_TrainingSet.size()-1);

//Milyen messze legyen egymástól két vektor, hogy nagyon különbözőnek
//számítson
int HammingLimit;
```

```
HammingLimit = 0.7 * m_TrainingSet[0].size();
vector<vector<int> > MostDifferent;

MostDifferent.push_back(m_TrainingSet[ThisVector]);

for(int n = 0; n < MostDifferent.size(); n++)
{
    if(n % 1000 == 0) cout << n << endl;
    for(int i = 0; i < m_TrainingSet.size(); i++)
    {
        if(HammingDistance(MostDifferent[n], m_TrainingSet[i]) >=
           HammingLimit)
        {
            if(std::find(MostDifferent.begin(), MostDifferent.end(),
                 m_TrainingSet[i]) == MostDifferent.end())
            {
                MostDifferent.push_back(m_TrainingSet[i]);
            }
        }
    }
}


for(int i = 0; i < MostDifferent.size(); i++)
{
    for(int j = i + 1; j < MostDifferent.size(); j++)
    {
        if(HammingDistance(MostDifferent[i], MostDifferent[j]) <
           HammingLimit)
        {
            MostDifferent.erase(MostDifferent.begin() + j);
            j--;
        }
    }
}
```

4.4.5.4 Tesztek, eredmények és értékelés.

Négyklaszteres 16 bites saját adathalmazra, kb 5 perc lefutás után:



11. ábra: Kohonen háló saját verziójának eredménye a 4 klaszteres 16 bites adathalmazon

Itt már végre az látható, amit eredetileg is látni szerettem volna, négy klaszter, egymástól messze. Az architektúra működőképes, az egyetlen probléma a három fázis arányának megválasztása, amit kikísérleteztem.

4.4.6 Tesztelés a végső verzióval

A következőekben be fogom mutatni a háló utolsó verzióján a tesztelést, először viszont pár szót szeretnék ejteni a tesztek validásáról

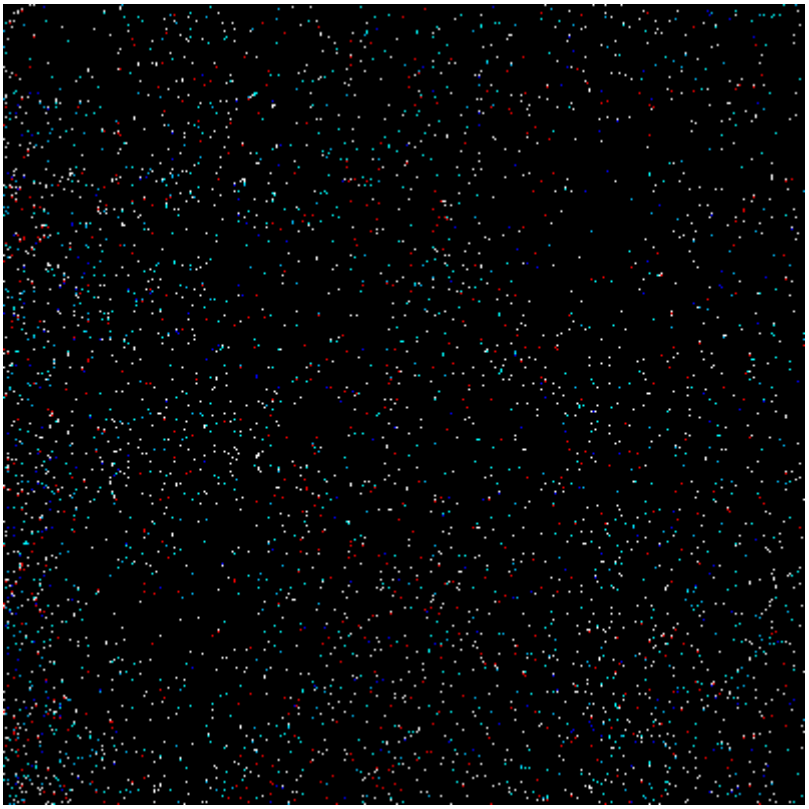
4.4.6.1 Validáció

A képek megtekintése közben felmerül a következő kérdés: „Odáig jó, hogy klaszterek vannak a képek, és tegyük fel, elhiszem, hogy ezek nem painttel készültek, hanem egy program kreálta őket, de honnan tudjuk, hogy tényleg a közeli vektorokat rakja-e egymás mellé? A képen nincs odaírva a vektor”.

Ez teljesen jogos kérdés, amire van válaszom, de csak a kisebb esetekre, ugyanis a nagyobb esetekre a validáció kézzel való elvégzése szinte lehetetlen, az automatizálására pedig nem volt idő. A kis tesztesetek validációja a következőként történt: A program egy

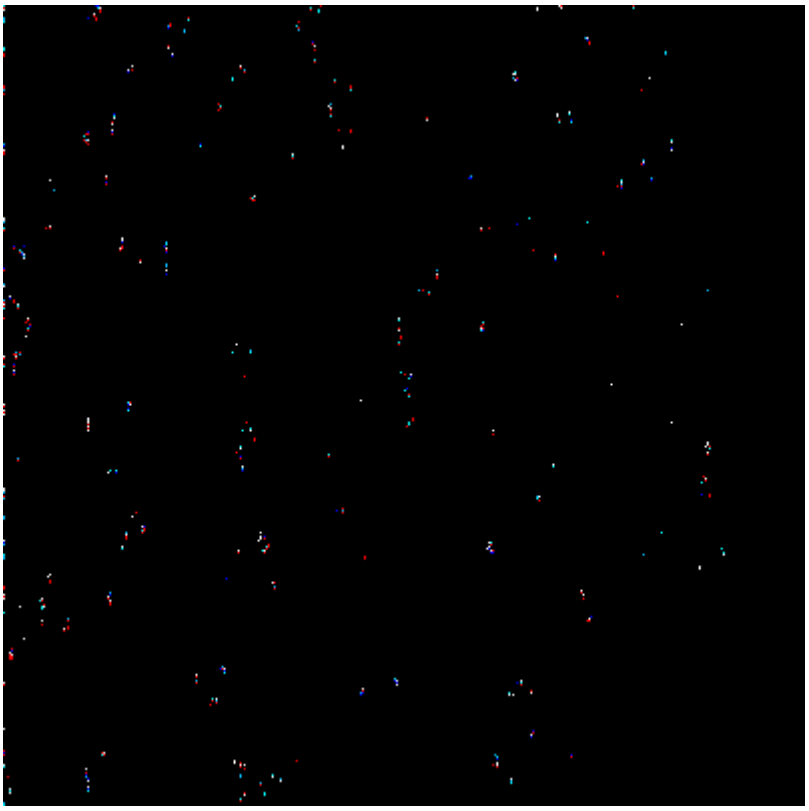
extra output file-t adott ki a vektorokkal és a hozzátartozó koordinátákkal, ezeket megnézve a kisebb eseteken tényleg a hasonlóak voltak egymás mellett.

4.4.6.2 PRNG



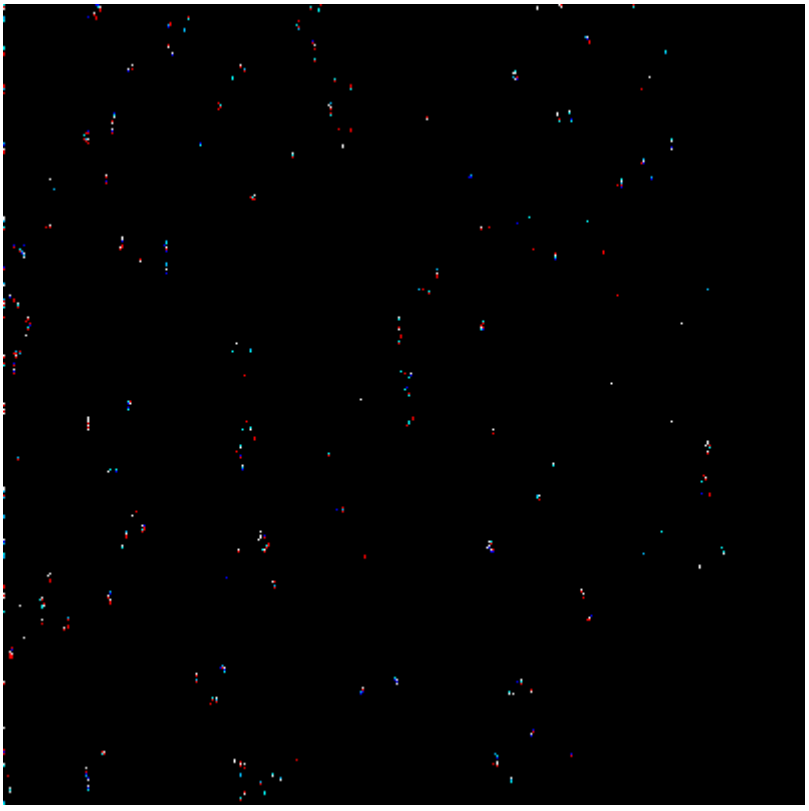
12. ábra: A végső Kohonen háló verziójának első fázisának lefutása a c++15 bites adathalmazon

Nem sok különbség látszik az eredeti próbálkozáshoz képest azon kívül, hogy színes, de ez még csak az első fázis lefutása, amely az eredeti klasztereket alakítja ki, ami szerintem egészen jól látszódik a fekete részeken, ahol a klaszterközpontok kitolják a nem hasonló adatokat.



13. ábra: Kohonen háló végső verziójának második fázisának lefutása c++ 15 bites adathalmazon

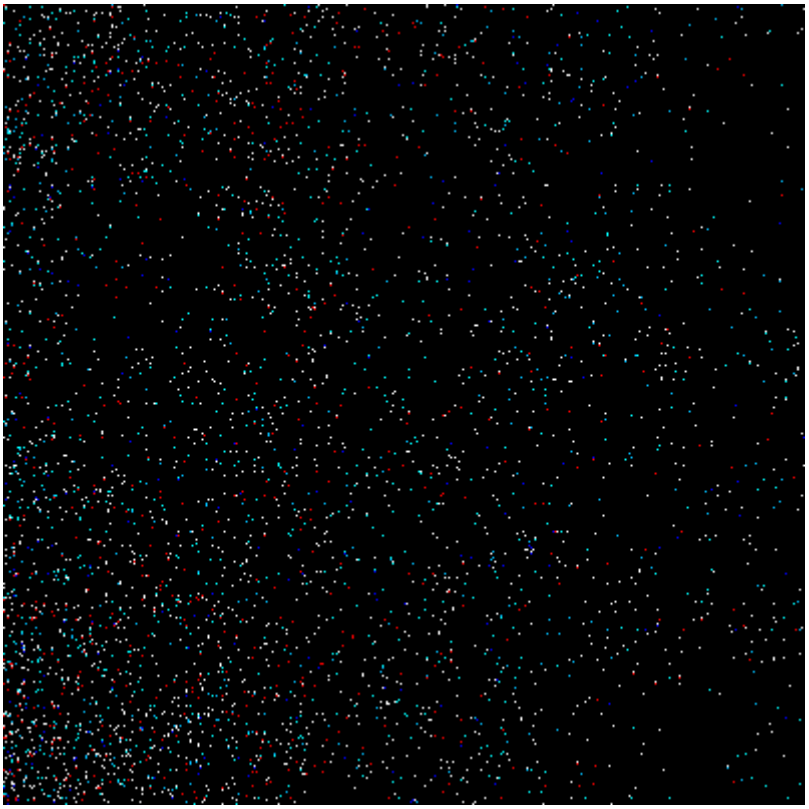
Ahogy reméltem látszanak is a klaszterek.



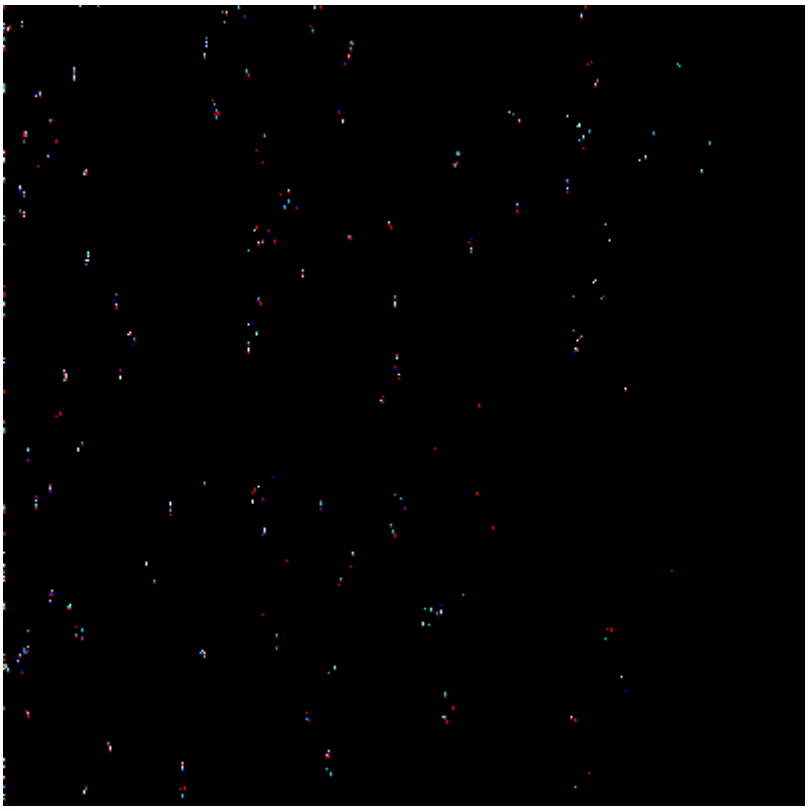
14. ábra: A Kohonen háló végső verziójának harmadik fázisának eredménye a c++ 15 bites adathalmazon

Az 11. ábrán nem sok változás látszik az előzőhöz képest, de ez leginkább azért van, mert idő szűkében nem maradt időm sokszor lefuttatni a rendkívül időigényes harmadik fázist.

A továbbiakban egyszer sem maradt időm lefuttatni a harmadik fázist, de a TDK követően még szeretnék tovább foglalkozni vele, pontosan milyen eredménye lenne

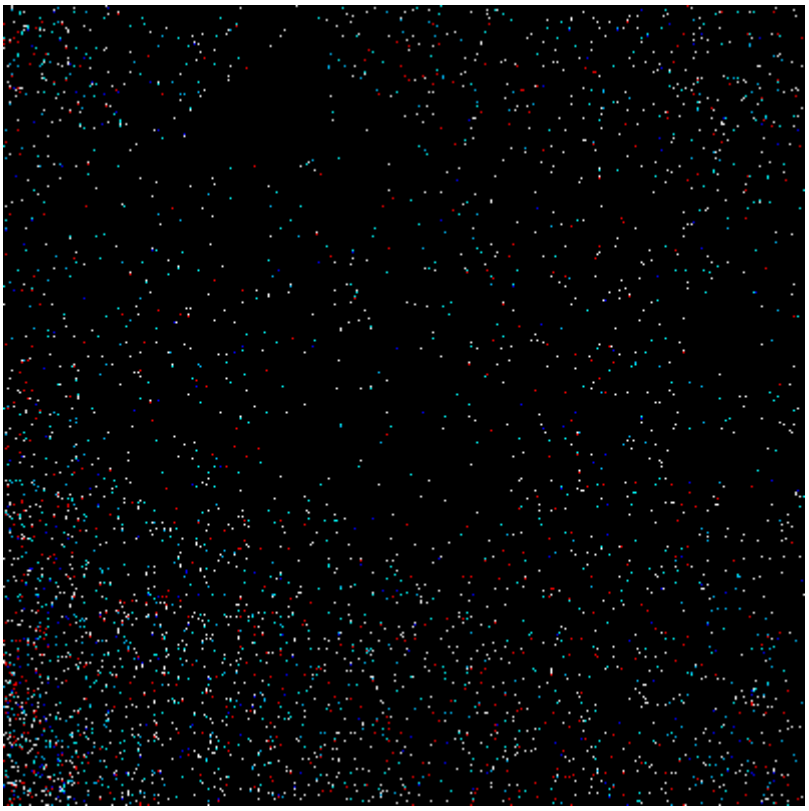


15. ábra: Kohonen háló, első fázis, MT19937 15 bites adathalmaz

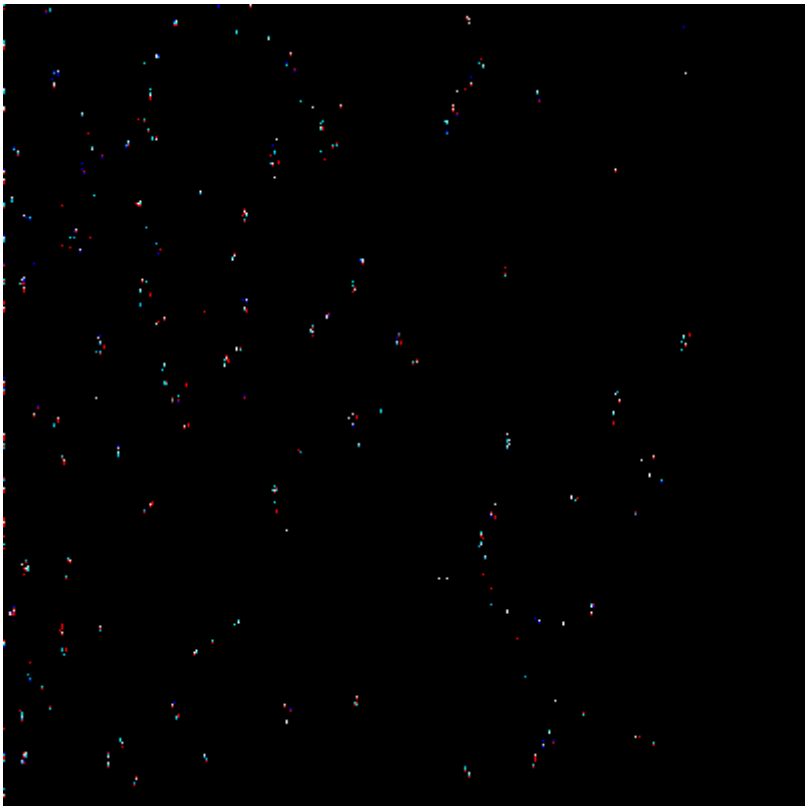


16. ábra: Kohonen háló második fázis MT19937 15 bites adathalmaz

4.4.6.3 TRNG

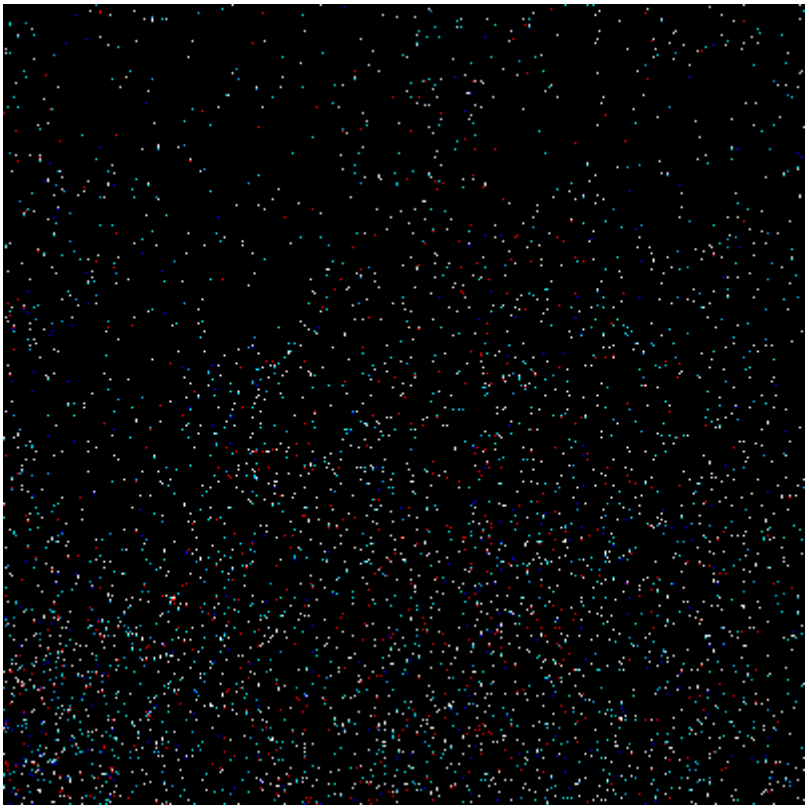


17. ábra, Kohonen háló lefutásának első fázisa a Random.org 15-bites adathalmazon

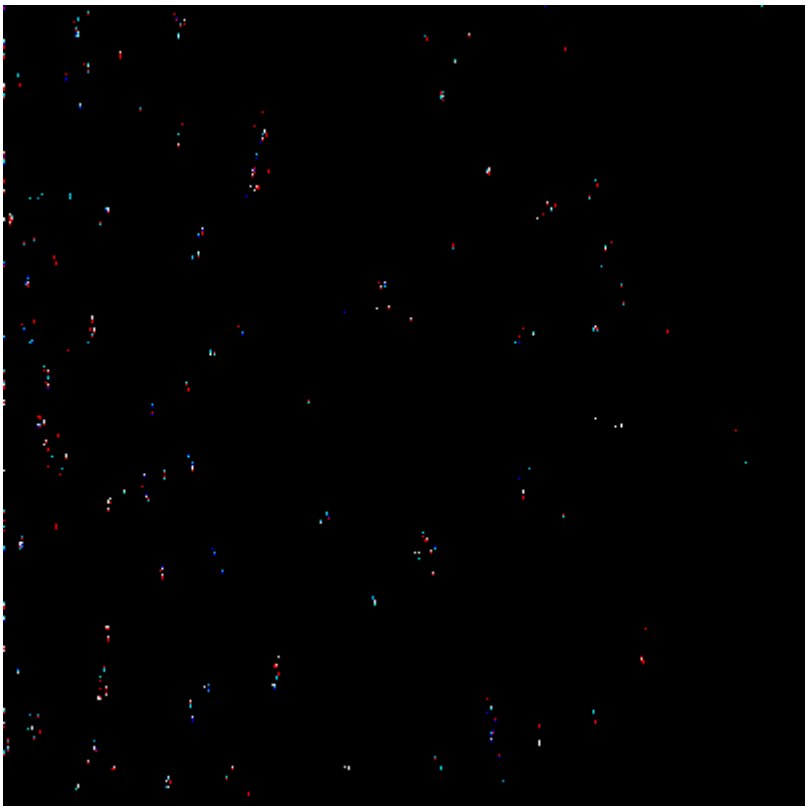


18. ábra: Kohonen háló végső verziójának második fázisa a Random.org 15-bites adathalmazon

4.4.6.4 QRNG



19. ábra: Kohonen háló végső verziójának első fázisa az ANU 15-bites adathalmazon



20. ábra: Kohonen háló végső verziójának második fázisa az ANU 15-bites adathalmazon

4.4.6.5 Összegzés és vélemény

Mielőtt elkezdeném az összegzést, néhány dolgot ki kell kötnöm: A tesztelés statisztikai jellegéből eredően, ezeket a generálásokat egy adatforrás nagyon sok adathalmazára kellene elvégezni, olyan módon, hogy minden adathalmaz más seedről/fizikai/kvantumos jelenségből eredjen. Az határidők és a háló hosszú futási ideje miatt én mindegyik adatforrásra csak egyszer tudtam lefuttatni, így az eredmények csak amolyan útmutatók, de nem jellemzik az egyes adatforrásokat, viszont egy kezdeti ötletet adnak a módszer érdeméről.

A képeken szépen látszódnak a klaszterek. Az elemzést többféleképpen is el lehet végezni, aminek a jelentősége, és a helyesen elvégzése a dolgozat alatt többször is hangsúlyozva volt.

Az egyik módszer, amivel jellemezhetjük a képeket egy egyszerű heurisztika, mekkora klaszterek vannak, milyen közel vannak egymáshoz, mennyi klaszter van. Ezzel a megközelítéssel egy probléma van: nem tudjuk, melyik kérdésre mi a jó válasz. Jó ha sok klaszter van vagy rossz? A következőekben megpróbálok egy valószínűségszámítási elemzési módszert adni a klaszterezésre:

Véletlenszámoknál az az optimális, ha a számot generáló eloszlás egyenletes, ezért ezt próbálom meg leképezni klaszterekre.

Jelölések:

- v – egy vektor
- n – vektor dimenziója
- N – vektorok száma
- P_{Hi}- Két vektor közötti i hamming távolság valószínűsége

Mivel egyenletes eloszlást feltételezünk, ezért egy adot pozícióban lévő bit értéke 0.5 valószínűséggel 0 és 0.5 valószínűséggel egy.

A Hamming távolság a két vektorban az azonos pozíciókban lévő eltérő bitek száma. Tehát akkor nő egyel az értéke ha két n dimenziós vektorban (v_1,v_2) azonos i pozícióban különböző bit van. Ennek a valószínűsége 0.5, mivel a 4 lehetséges esetből 2-ben teljesül ez.

Tehát $P_{Hi}= 0.5^i$

A tesztjeim esetében: N ~ 15 000 és n = 15

A Hamming határom pedig ennek a héttizede. Azaz 10.

Ekkor a kezdő klaszterek száma max 0.5^{11}*150000 ~ 7. A többit azért nem nézzük meg mert a magasabb Hamming távolsággal rendelkező vektorokból kevesebb van, és ha egyet be akarnánk venni ezek közé, annak az első 11-bitje hasonlítana az előzőekre.

Viszont mivel a heurisztikám csak elég jót keres ezt nem mindig bizotsítja, ennél kevesebb lesz, viszont a későbbi lépések ezt kiegyenlítik. Az eredmények nagyjából ezt a számot is közelítik. Ekkor a többinek is ilyen esélyeik vannak és ekörül a hét vektor körül kell gyűlniük. A hét vektor körül nagyjából ugyanannyi vektornak kell lennie, és a klasztereknek nem szabad túl messze lenni egymástól.

A képek elemzése:

A c++ képen nagyjából 7 kaszter alakult ki, viszont nagyon észrevehetően elhatárolódnak a klaszterek egymástól, és szét vannak egymáshoz képest is szórva. A Random.org és ANU-s adathalmazban vannak nagy klaszterek amik összeérnek, viszont tartalmaznak kiugró klasztereket is. Ezek alapján is látszik, hogy a c++ generátora egy kicsit a másik kettő alá esik minőségileg. Ennek ellenére még egyzer kikötném, hogy még sokszor le kellene futtatni a teszteket, hogy rendesen össze lehessen hasonlítani a generátorokat.

4.5 Konvolúciós neurális hálózat (CNN)

Azt, hogy a CNN-t használni fogom már nyilvánvaló volt, amikor neurális hálózatokról gondolkoztam a tesztelés során. A CNN-eket általában képfelismerésre és osztályozásra szokták használni, én az utóbbira fogom, az adathalmazaimból képeket fogok generálni, és megnézem, a háló képes-e megállapítani, melyik kép melyik generátortól származik.

4.5.1 Elmélet és háttér

A konvolúciós neurális hálózatok alapötlete egy 1968-as tanulmányból[8] származik, melyben majom szemeket vizsgáltak, és felfedezték, hogy a majom és macskaszemben neuronok vannak, melyek egyenként felelnek a látómező egy részéért. Ha a szemek nem mozognak, a látómező azon részét, amin belül a vizuális stimuláció befolásolja egy neuron tüzelését, a neuron receptív mezejének nevezzük. Szomszédos cellák hasonló, és egymást fedő receptív mezőkkel rendelkeznek. Ezt a rendszert modellezi a konvolúciós neurális hálózat.

A konvolúciós hálók rétegekből állnak. Alapvetően egy szimpla CNN-ben háromfajta réteget különböztetünk meg, a konvolúciós réteget, a pooling réteget és a fully-connected réteget. Mind a háromnak más funkciója van a hálózaton belül.

A konvolúciós rétegek dolga a megkapott mátrixot végigpásztázni és a pásztázott részeket konvulálni, így tömöríteni az ott látott adatokat és tovább adni a következő rétegnek egy aktivációs függvényen keresztül

A pooling rétegek feladata a megkapott mátrixot mintavételezni , így tömöríteni az onnan kapott adatokat.

A Fully-Connencted(FC) réteg feladata pedig a kapott összetömörített adat alapján eldönteni, hogy az eredeti kép melyik osztályba tartozik.

A pooling és konvolúciós rétegek általában valamilyen aránnyal folyamatosan váltakoznak az architektúrán belül, a hálózat méretétől függően, de az FC réteg a legtöbb architektúrában a hálózat legutolsó/output rétege.

A konvolúciós hálózatoknak még rengeteg paramétere lehet, mint például a drop-out, az optimalizációs függvény, így a részletes bemutatását nem írom le, mert az is egy önálló dolgozat lenne.

4.5.2 Implementáció

A háló implementációja egy gyors folyamat volt hála a tensorflow segédfüggvényeinek és struktúráinak. A munka neheze a képeim rávitele a hálóra volt. Több különböző error-t is kaptam a fejlesztés során melyek közül a legbeszédesebbek a: „Kernel died, restarting” és az

„Iteration stopped” volt, a másodiknak az okára máig sem tudtam rájönni, bár megoldani meg tudtam egy mappa átnevezésével, és egy program újraindításával.

4.5.3 Tesztelési adatok

A teszteléshez a számokat át kellett alakítani képekké, hogy egy általános konvolúciós hálózat működjön rajta. Itt egy problémába ütköztem, amit nem teljesen tudtam megoldani. Ahogy az adathalmazokat részletező fejezetben leírtam. A legtöbb generátor csak limitált bitmennyiséget tud generálni egyben: a legnagyobb az összehasonlításhoz az 30 bit volt amiből csak 5*5 ös képeket lehet kinyerni, ami egy konvolúciós hálózatnak nem elég. A másik lehetőség az volt, hogy összefűzöm a 30 bites számokat képekké és azokat elemzem, de ennél a módszernél nem vagyok biztos benne, hogy a generátor eloszlását helyesen reprezentálom a képekkel. Végül mind a két módszert kipróbáltam, a második módszernél 60*60-as képeket generáltam.

Néhány kép az átalakítottak közül:

 60*60-as kép az ANU generátorától

 60*60-as kép a Mersenne Twistertől

 60*60-as kép a Random.org-ról

 5*5-ös kép az ANU generátorától

 5*5-ös kép a Mersenne Twistertől

 5*5-ös kép a Random.org-ról

4.5.4 Tesztelési eredmények

A fenti képeken nem sok megkülönböztethető formát látunk. Ez többek között azért is van mert nem az összes képet tettem be, mert akkor még 400 oldal hosszú lenne a dolgozat minimum. De nézzük meg, hogy a konvolúciós hálónk sikerrel járt-e a megkülönböztetésben:

```
Step 7500, Minibatch Loss= 1.1041, Training Accuracy= 0.320
Step 7600, Minibatch Loss= 1.1015, Training Accuracy= 0.312
Step 7700, Minibatch Loss= 1.1037, Training Accuracy= 0.234
Step 7800, Minibatch Loss= 1.0973, Training Accuracy= 0.391
Step 7900, Minibatch Loss= 1.1000, Training Accuracy= 0.297
Step 8000, Minibatch Loss= 1.0992, Training Accuracy= 0.320
Step 8100, Minibatch Loss= 1.1032, Training Accuracy= 0.227
Step 8200, Minibatch Loss= 1.0956, Training Accuracy= 0.398
Step 8300, Minibatch Loss= 1.1036, Training Accuracy= 0.258
Step 8400, Minibatch Loss= 1.1004, Training Accuracy= 0.375
Step 8500, Minibatch Loss= 1.1013, Training Accuracy= 0.305
Step 8600, Minibatch Loss= 1.1002, Training Accuracy= 0.281
Step 8700, Minibatch Loss= 1.1027, Training Accuracy= 0.289
Step 8800, Minibatch Loss= 1.1040, Training Accuracy= 0.320
Step 8900, Minibatch Loss= 1.0968, Training Accuracy= 0.273
Step 9000, Minibatch Loss= 1.1012, Training Accuracy= 0.289
Step 9100, Minibatch Loss= 1.0937, Training Accuracy= 0.367
Step 9200, Minibatch Loss= 1.1015, Training Accuracy= 0.281
Step 9300, Minibatch Loss= 1.0903, Training Accuracy= 0.352
Step 9400, Minibatch Loss= 1.0941, Training Accuracy= 0.383
Step 9500, Minibatch Loss= 1.0995, Training Accuracy= 0.375
Step 9600, Minibatch Loss= 1.1022, Training Accuracy= 0.328
Step 9700, Minibatch Loss= 1.0961, Training Accuracy= 0.406
Step 9800, Minibatch Loss= 1.0962, Training Accuracy= 0.328
Step 9900, Minibatch Loss= 1.0971, Training Accuracy= 0.367
Step 10000, Minibatch Loss= 1.1022, Training Accuracy= 0.258
Optimization Finished!
```

21. ábra: 5*5-ös képeken

```
Step 7600, Minibatch Loss= 1.0942, Training Accuracy= 0.422
Step 7700, Minibatch Loss= 1.0983, Training Accuracy= 0.312
Step 7800, Minibatch Loss= 1.1041, Training Accuracy= 0.312
Step 7900, Minibatch Loss= 1.0958, Training Accuracy= 0.359
Step 8000, Minibatch Loss= 1.0985, Training Accuracy= 0.328
Step 8100, Minibatch Loss= 1.1046, Training Accuracy= 0.266
Step 8200, Minibatch Loss= 1.1051, Training Accuracy= 0.336
Step 8300, Minibatch Loss= 1.0972, Training Accuracy= 0.344
Step 8400, Minibatch Loss= 1.0996, Training Accuracy= 0.352
Step 8500, Minibatch Loss= 1.1005, Training Accuracy= 0.312
Step 8600, Minibatch Loss= 1.0996, Training Accuracy= 0.320
Step 8700, Minibatch Loss= 1.0999, Training Accuracy= 0.352
Step 8800, Minibatch Loss= 1.0987, Training Accuracy= 0.297
Step 8900, Minibatch Loss= 1.0937, Training Accuracy= 0.305
Step 9000, Minibatch Loss= 1.1000, Training Accuracy= 0.336
Step 9100, Minibatch Loss= 1.0981, Training Accuracy= 0.422
Step 9200, Minibatch Loss= 1.0993, Training Accuracy= 0.305
Step 9300, Minibatch Loss= 1.0977, Training Accuracy= 0.336
Step 9400, Minibatch Loss= 1.1018, Training Accuracy= 0.305
Step 9500, Minibatch Loss= 1.0973, Training Accuracy= 0.352
Step 9600, Minibatch Loss= 1.1025, Training Accuracy= 0.289
Step 9700, Minibatch Loss= 1.0954, Training Accuracy= 0.336
Step 9800, Minibatch Loss= 1.0997, Training Accuracy= 0.375
Step 9900, Minibatch Loss= 1.1003, Training Accuracy= 0.352
Step 10000, Minibatch Loss= 1.1052, Training Accuracy= 0.359
Optimization Finished!
ERROR:tensorflow:Exception in QueueRunner: Enqueue operation was ca
```

22. ábra: 60*60-as képeken

Ahogy látható a pontosságból (0 és 1 közötti érték) a hálónak nem sikerült megtanulnia a megkülönböztetést. Mivel három osztály volt (Mersenne Twister, Random.org, ANUQRNG) az átlag pontosság pedig 3.5 ezért azt tudjuk levonni, hogy a háló egy picit jobban teljesített, minthogyha véletlenül tippelt volna, viszont ez sokkal tovább tartott neki. Bár ez a kísérlet bukásnak tűnik, és valószínűleg nem is lehetséges módja a tesztelésnek, azt ki kell kötnöm, hogy a konvolúciós háló amit használtam nagyon egyszerű volt, és a tanítási iterációkkal sem volt időm kísérletezni, de a legkorlátozóbb változó a generálható bitmennyiség volt. A konvolúciós hálók nagy képeken működnek jól, és általában akkor mintákat keresnek, mint amekkorák az én képeim voltak, ami ezért nem volt valami hatásos.

A konvolúciós hálózatoknak, mint a legtöbb neurális hálózatnak több adathalmaz szokott kelleni, tanító, validáló, tesztelő. Az esetemben mivel a tanító mintákra sem tudott rátanulni, az utóbbi kettőt kihagytam.

Összegzés

A dolgozatban egy rövid elméleti áttekintés után bemutattam az adathalmazaimat amiket generáltam, majd a három modellemet, MLP, Kohonen háló és CNN. A modellek bemutatása során kitértem a fejlesztések fázisaira, a tesztekre amiket futattam, ezek eredményeit és a róluk levont következtetéseket. A dolgozat megírása egy kihívás volt a témák nehézsége, és a terület újdonsága miatt, de élvezetes volt olyan dolgokon dolgozni amivel mások még nem igazán foglalkoztal. Ezek az ötletek tovább finomíthaóak és új modellekkel is ki lehet igazítani. Ha sikerül felhasználni valaha a neurális hálózatok összes képességét a véletlenszám-generátor tesztelésében, az hatalmas jelentősségel bírna a véletlenszám alapú biztonsági rendszerek fejlődésében.

Mindent egybevéve a végére, bár nem lettek feltétlenül látványos eredményeim, a munka megérte. Az eredményeim alátámasztják, hogy nagy potenciál van a neurális hálózatok felhasználására a véletlenszámok tesztelésének témakörében, bár az egzakt architektúrákon és paraméterezéseken még dolgozni kell, ami a téma jellegzetességét tekintve igen időigényes, mivel a tesztek statisztikussága miatt sokszor kell őket futatni, a neurális hálózatok miatt pedig egy futás nagyon sok ideig tart.

Köszönetnyilvánítás

A TDK-dolgozatban megvalósuló kutatás a "Kvantumbitek előállítása, megosztása és kvantuminformációs hálózatok fejlesztése" nevű, 2017-1.2.1-NKP-2017-00001 számú projekt támogatásával készült, amely projekt a Nemzeti Kutatási Fejlesztési és Innovációs Alapból biztosított támogatással, a "Nemzeti kiválósági program" pályázati program finanszírozásában valósul meg.

Irodalomjegyzék

[1] Imre S., Balázs F., „Quantum Computing and Communications, An Engineering Approach, Wiley, 2004

[2] Sheng-Kai Liao et al „Satellite-relayed intercontinental quantum network”, 2018 (Utolsó látogatás, 2018.10.24)

[3] https://www.dwavesys.com, *(Utolsó látogatás, 2018.10.24)*

[4] http://www.mcl.hu *(Utolsó látogatás, 2018.10.24)*

[5] ANU: *Quantum Random Numbers Server*, http://qrng.anu.edu.au *(Utolsó látogatás dátma: 2018. október 24.)*

[6] Marvin Minsky, Seymour A. Papert, „Perceptrons”, könyv, 1969, MIT nyomda

[7] Altrichter Márta, Horváth Gábor, Pataki Béla, Strausz György, Takács Gábor, Valyon József, „Neurális Hálózatok”, 2006, Felsőoktatási Tankönyv- és Szakkönyv-támogatási Pályázat

[8] David H. Hubel, Torsten Wiesel „Receptive fields and functional architecture of monkey striate cortex” (1968) The Journal of Physiology. 195 (1): 215–243.

[9] LeCun, Yann; Bengio, Yoshua (1995). "Convolutional networks for images, speech, and time series". „*The handbook of brain theory and neural networks”* című könyv 255-258 oldal

[10] Andrew Rukhin et al, „A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”, Revisited: 2010, a weboldalukon jelenítették meg

[11] Makoto Matsumoto, Takuji Nishimura, „Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator” 1998, ACM Transactions on Modelling and Computer Simulation (TOMACS) külön kiadás. 8.kötet 1. kiadvány

[12] random.org weboldal, *(Utolsó látogatás dátma: 2018. október 24.)*

[13] quantumrandom: *Project by Luke Macken (Utolsó látogatás dátma: 2018. október 24.)*

[14] https://www.eclipse.org/oxygen : Java fejlesztő környezet (*Utolsó látogatás dátuma: 2018:10:28*)

[15] http://www.codeblocks.org/ : c++ fejlesztő környezet (*Utolsó látogatás dátuma: 2018:10:28*)

[16] https://www.anaconda.com/ : Python fejlesztői platform (*Utolsó látogatás dátuma: 2018:10:28*)

[17] https://www.tensorflow.org : Python nyelvkiegészítő (*Utolsó látogatás dátuma: 2018:10:28*)

[18] https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py, eredeti MLP github projekt (*Utolsó látogatás dátuma: 2018:10:28*)

[19] https://codesachin.wordpress.com/2015/11/28/self-organizing-maps-with-googles-tensorflow/ : tutorial a tensorflow-os SOM-hoz (*Utolsó látogatás dátuma: 2018:10:28*)

[20] http://www.ai-junkie.com/ann/som/som1.html : tutorial a c++-os SOM-hoz (*Utolsó látogatás dátuma: 2018:10:28*)

[21] Mustapha Lebbah, Younés Bennani, Nicoleta Rogovschi, „A Probabilistic self-organizing map for binary data topographic clustering” 2007, revised 2008, International Joint Conferences on Neural Networks, IJCNN 2007 Agusztus 12-17, Orlando, Florida, USA

[22] Mustapha Lebbah, Fouad Badran, Sylvie Thiria, „Topological Map for Binary Data”, 2000, Europan Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), 2000 Április 26-28, Bruges, Belgium

Függelék

A következő mintahalmazokat kézzel, vektoronként és bitenként alkottam meg, nem generátorral, ezért teszem bele a függelékbe.

Kohonen háló saját adathalmazok:

10 bites:

[1111100000] [0000011111]

[0111100000] [0000001111]

[1011100000] [0000010111]

[1101100000] [0000011011]

[1110100000] [0000011101]

[1111000000] [0000011110]

16 bites:

[1111000000000000][0000111100000000][0000000011110000][0000000000001111]

[0111000000000000][0000111000000000][0000000001110000][0000000000000111]

[1011000000000000][0000101100000000][0000000010110000][0000000000001011]

[1101000000000000][0000110100000000][0000000011010000][0000000000001101]

[1110000000000000][0000111000000000][0000000011100000][0000000000001110]

MLP saját adathalmazok:

Az utolsó bit minden vektorban a megoldása a feladatban

XOR (az AND ugyanez csak az eredmény van megcserélve):

Teszt	Validáció
110110	100010
110001	011011
111100	011100
011001	100111
110100	111110
111001	101101
010110	000000

010011	111011
111110	101010
110011	000101
100000	
101111	
001000	
001111	
001010	
101101	
101000	
100101	
101010	
001101	

Andor(AndXOR)

Teszt	Validáció
11100110001	11111111001
01011000000	11111110011
10111110101	10100111101
01000001110	10111110011
10100110101	11010101111
01001000100	10000100011
11111110101	10100101101
01011001110	11100101011
10100111111	00100010011
00000000000	01101010111
10111110001	10100010011

01000001010	00100110011
11100110101	00000000010
00011001110	00000000100
10100111011	10000000000
01001001010	11100000000
10111110011	10100000000
01000001010	10111000000
11111111111	00000111110
10100000010	00000111010
11010100001	00000110100
00011001100	11000010010
10100110001	01110100000
11111001110	01100101100
11001110101	
10110001010	
11010110001	
10111001110	
10010100001	
11100000010	
10100100001	
10111001010	
11100101001	
10111001110	
10000101101	
10100001110	
10000100001	
11111000000	

10110100001	
11111001010	
11100101111	
00010101010	
01100010001	
00011101000	
01111010001	
01001101110	
01101110001	
01010110110	
00111110001	
01010110010	
10101010001	
01111101110	
00100011001	
01111101010	
00101110101	
11011100010	
01101111101	
11000101110	
10101010001	
10011101110	
01111111111	
00101011111	