[Külső fekete borítólap formátuma]

Széchenyi István Egyetem

Gépészmérnöki, Informatikai és Villamosmérnöki Kar

Informatika Tanszék

**SZAKDOLGOZAT**

**Németh Valentin**

**Mérnök Informatikus BSc**

2021

|  |
| --- |
| Németh Valentin, 2021 |

**SZAKDOLGOZ****AT**

**Kotta digitalizáló szoftver fejlesztése**

**Németh Valentin**

**Mérnök Informatikus BSc**

**2021**

# Nyilatkozat

Alulírott, Németh Valentin (KNPDXS) Mérnök Informatikus BSc szakos hallgató kijelentem, hogy a Kotta digitalizáló szoftver fejlesztése című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, [dátum]

hallgató

# Kivonat

Kotta digitalizáló szoftver fejlesztése

A szakdolgozat célja, hogy elkészítsek egy zenei kotta digitalizálását megvalósító szoftvert. A szoftver bemenetként kap egy képet egy zenei kottáról, amelyen a hangjegyek és egyéb szimbólumok találhatóak. A program felismeri ezeket, valamint az elhelyezkedésüket is, ez alapján feldolgozza a képet, melyet egy olyan kimenetté alakít át, amelyet egy kottaolvasásra alkalmas program is értelmezni tud bemenetként.

A dolgozat első felében a szoftver fejlesztésének elméleti hátterével foglalkozok részletesebben. Ebben a részben ismertetem a legfontosabb alapokat, melyek a szoftver tényleges fejlesztéséhez elengedhetetlenek. Előszőr inkább általánosan, a használt programnyelvről és a megvalósításhoz szükséges információkról lesz szó, majd utána specifikusabban a szakdolgozathoz szükséges elemekről.

A diplomamunka második felében bemutatom, hogyan valósítottam meg a szoftver elkészítését. Itt már főként a gyakorlati megoldások lesznek bemutatva, a használt programkódokkal alátámasztva. Lépésről lépésre követhető lesz a programozás menete, s mellette folyamatos magyarázatok segítik az olvasót annak megértésében, hogy a feladatot miért az adott metódusokkal oldottam meg. A megvalósítás végén keletkező kimenetet is bemutatom, valamint különböző tesztelésekre is sor fog kerülni.

A szakdolgozat elkészítéséhez különböző programozási ismeretek, valamint a gépi látás és az azt támogató programcsomagok szükségesek. A megvalósítás során sok új ismeretre tettem szert, amelyek az élet során fontos tudást jelenthetnek számomra, amennyiben a jövőben szoftverfejlesztési feladatokkal lesz dolgom.

# Abstract

Developing a sheet music digitizer software

The aim of the dissertation is to create a software for digitizink a sheet music. The software gets a sheet music as an input, that includes notes and other symbols. The program recognizes these as well as their location, it processes the image, which it converts into an output that can also be interpretes an an input by a program that is capable of reading sheet music.

In the first half of the dissertation i deal with the theoretical background of software development in more detail. In this section, I describe the most important basics, that are essential for the actual development of the software. First, I will be more general, with the presentation of the programmin language used and the information needed for the implementation, then the more specific elements, whicg are needed for the dissertation.

In the second half of the dissertation I will show you how I implemented the software. Here, mainly the practical solutions will be presented, supported by the program codes used. The programming process will be followed step by step, and continuous explanations will help the reader to understand why I solved the task with the given methods. I will also present the output at the end of the implementation, and various test will take place.

The preparation of the dissertation requires various programming skils, as well as machine vision and the software packages that support it. During the implementation, I gained a lot of new knowledge, which can be an important knowledge for the rest of my life, if I have to deal with software development tasks in the future.

# Tartalomjegyzék

Tartalom

[Nyilatkozat 3](#_Toc99269805)

[Kivonat 4](#_Toc99269806)

[Abstract 5](#_Toc99269807)

[Tartalomjegyzék 6](#_Toc99269808)

[1. Bevezetés 8](#_Toc99269809)

[2. A programozási nyelv 9](#_Toc99269810)

[2.1 A nyelvek fejlődése 9](#_Toc99269811)

[2.2 A programozási nyelvek generációi 9](#_Toc99269812)

[2.3 Imperatív és Deklaratív nyelvek 11](#_Toc99269813)

[2.4 Eljárásorientált és Objektumorientált nyelvek 11](#_Toc99269814)

[3. A Python 12](#_Toc99269815)

[3.1 Története 12](#_Toc99269816)

[3.2 A Python néhány fontos jellemzője: 13](#_Toc99269817)

[3.3 További tulajdonságok 14](#_Toc99269818)

[3.4 Milyen programok írásához jó? 14](#_Toc99269819)

[4. Gépi látás 16](#_Toc99269820)

[4.1 Alkalmazási területei 16](#_Toc99269821)

[5. A program megtervezése 18](#_Toc99269822)

[5.1 Kép előfeldolgozása 19](#_Toc99269823)

[5.2 Vonalkeresés 21](#_Toc99269824)

[5.2.1 Hough Line Transform 21](#_Toc99269825)

[5.2.2 Probabilistic Hough Transform 23](#_Toc99269826)

[5.2.3 Morfológiai műveletek 25](#_Toc99269827)

[5.3 Kontúrkeresés 31](#_Toc99269828)

[5.4 Konvolúciós neurális hálózat 33](#_Toc99269829)

[5.4.1 A hagyományos CNN szerkezete 35](#_Toc99269830)

[5.4.2 Objektum felismerés a CNN használatával 36](#_Toc99269831)

[5.5 Régiókon alapuló konvolúciós neurális hálózat 37](#_Toc99269832)

[6. A program megvalósítása 39](#_Toc99269833)

[6.1 Előkészületek 40](#_Toc99269834)

[6.2 Küszöbérték 40](#_Toc99269835)

[6.3 Kottavonalak megkeresése 40](#_Toc99269836)

[6.4 Kottavonalak kitörlése 42](#_Toc99269837)

[6.5 Alakzatok kontúrjainak megkeresése és megjelölése 43](#_Toc99269838)

[6.6 Alakzatok eltárolása 46](#_Toc99269839)

[6.7 Neurális modell létrehozása 49](#_Toc99269840)

[6.8 Neurális modell tesztelése 50](#_Toc99269841)

[6.9 Neurális modell implementálása a programba 50](#_Toc99269842)

[Irodalomjegyzék 52](#_Toc99269843)

# Bevezetés

Szakdolgozatom célja, hogy elkészítsek egy zenei kotta digitalizálására alkalmas szoftvert. A megvalósítás során a fejlesztés lépéseit és az elméleti hátterét is dokumentálom. A szoftver Python programozási nyelvben lesz fejlesztve, különböző gépi látást támogató csomagok segítségével. A program bemenetként egy képet kap a kottáról. A szoftvernek fel kell ismernie a zenei kotta meghatározott alkotóelemeit és azoknak elhelyezkedését, majd a felismert kész képből valamilyen digitalizált formában egy olyan kimenetet generál, amelyet egy kottaolvasó szoftver is képes felismerni. A megvalósítás során kiválasztom a megfelelő metódusokat a felismerni kívánt elemek definiálására, generálok egy olyan kimenetet, amelyet egy kottaolvasó szoftver is kezelni tud, valamint különböző teszteket készítek, amelyeken lefuttatom a programomat, hogy az esetleges hibákat ki tudjam küszöbölni. A tesztek különböző bemeneti képek lesznek, amelyeken a szoftver hatékonyságát is ellenőrzöm.

# A programozási nyelv

Minden programozási feladat első lépése meghatározni, hogy milyen környezetben akarjuk megvalósítani a fejlesztést. Ezt legnagyobb mértékben a választott programozási nyelv határozza meg. Manapság már sokféle lehetőségből választhat a fejlesztő, azonban ez nem mindig volt így.

## A nyelvek fejlődése

A programozás, mint fogalom az 1950-es években került először szóba, miután 1952-ben Neumann János elvei alapján elkészült az EDVAC, az első programozható számítógép. Itt még nem a mai értelemben vett programozást kell érteni, hiszen az akkori programok még csak matematikai problémák megoldására voltak alkalmasak. A kódok megírása és maga a kódnyelv is teljesen kezdetleges volt még abban az időben, a programozás tulajdonképpen nem volt más, mint a számítógépnek megadott utasítások sorozata. A programok „gép-függőek” voltak, azaz egy programot egy adott számítógépre lehetett csak megírni. A kódok megírása hosszú és körülményes munka volt. [4]

A 60-as, 70-es években sikerült gyorsabbá és egyszerűbbé tenni a programfejlesztési módszereket, és megszülettek az első magas szintű programozási nyelvek is. Az új programozási nyelveknek köszönhetően olyan módszerek és technikák alakultak ki, amelyek nagyban megkönnyítették a szoftverfejlesztők feladatát. Az újonnan kialakult nyelvek között voltak általános célúak, de voltak specializáltak is. [4]

A 80-as, 90-es években jelent meg az objektumorientált programozás, valamint a Script-nyelvek, amelyek ma is a leghasználatosabb módszerek. Egyes scriptnyelveket hibridnyelvnek is neveznek, mivel egyszerre van lehetőség bennük az eljárásorientált és az objektumorientált programozásra is. Ilyen például a PHP, a Javascript vagy az Actionscript. [4]

## A programozási nyelvek generációi

1. generáció:

* gépi kódban írták őket
* nehézkes és körülményes programírás
* erősen gépfüggő
* gyors lefutási idő
* a processzor közvetlenül végrehajtja az utasításokat
* az Assembly nyelv is ide tartozik, de az már szöveges kódú
* Assembler, azaz fordítóprogram szükséges hozzá
* kapcsolatban áll a gépi nyelvvel

1. generáció:

* megjelentek a magasszintű programnyelvek
* áttekinthetőbb a program
* egyszerűbb és gyorsabb programírás
* hosszabb lefutási idő a fordító miatt
* kevesebb hibalehetőség a fordítók fejlettsége miatt
* kialakulnak az alapvető utasítástípusok
* megszületnek az elemi adattípusok
* létrejönnek az első összetett adatszerkezetek, a tömb és a rekord
* pl. FORTRAN, ALGOL, COBOL, BASIC

1. generáció

* a nyelvek elkezdtek a problémák alapján specializálódni
* megjelent a procedurális szemlélet és a struktúrák hangsúlyozása
* megszülettek az általános célú, az objektumorientált és a specializált nyelvek
* az általános célúak közé tartozik a PASCAL, ADA, APL, C
* az objektumorientált nyelvekhez tartozik a C++, Turbo Pascal, Smalltalk
* a specializált nyelvek többek között a Logo, Forth

1. generáció:

* 4GL rendszerek
* ezek már kifejezett programnyelvek
* magasszintű programnyelvre épülő, komplex, objektumorientált fejlesztői környezet
* modularitás jellemzi
* adatbázissal szorosan kapcsolódnak
* pl. Visual Basic, Visual C++, Java, Object Pascal

[4]

## Imperatív és Deklaratív nyelvek

A programnyelveket a kód szerkezete alapján csoportosíthatjuk imperatív és deklaratív csoportba. Ezek néhány lényeges dologban térnek el egymástól.

Az imperatív nyelvek algoritmikus nyelvek, ahol a program főként utasításokból áll, és a változók a legfontosabb részei a programoknak. Itt a programozó határozza meg a feladat megoldását, és a tárkezelésről is ő gondoskodik. Ezeknek a programozási nyelveknek két alfajtája van: az eljárásorientált és az objektumorientált. [4]

A deklaratív nyelvek esetében a programozó csak a megoldandó feladatot deklarálja, de a megoldást nem ő dolgozza ki. Ugyan változók ennél a programnyelv típusnál is megtalálhatóak, jelentésük más, mint az imperatív nyelveknél. Hasonlóan a másik nyelvhez, itt is utasításokat ad a szoftverfejlesztő a számítógépnek, ám a memóriát nem tudja kezelni. A deklaratív nyelvek alfajtái a funkcionális és a logikai nyelvek. [4]

## Eljárásorientált és Objektumorientált nyelvek

Az eljárásorientált nyelvek egyik ismertető jele, hogy a megoldandó feladatot kisebb részegységekre, más néven eljárásokra bontja. Ezeket az alprogramokat jól körül kell határolni, mivel a fő program majd ezeket fogja sorozatosan meghívni. Amikor megtörténik a meghívás, az eljárás kap egy vagy több bizonyos paramétert, amelyet a benne meghatározott logika szerint valamilyen visszatérési értékké alakít át és ad vissza a fő programnak. A kettő legfőbb adattípusa az eljárásorientált programoknak a tömb és a rekord. [4]

Az objektumorientált programnyelvek - ahogy a nevükben is benne van – objektumokon alapulnak. Objektumnak hívjuk a programban szereplő adatokat és a hozzájuk tartozó műveleteket. Mindegyik objektum rendelkezik tulajdonságokkal és metódusokkal. Ezek a nyelvek osztályokra épülnek, és az objektumok az osztályok egy-egy példányai. Az osztályoknak lehetnek ősosztályaik, akiktől örökölhetnek tulajdonságokat, metódusokat, illetve lehetnek származtatott osztályaik, akiknek ők adhatnak tovább. Az öröklődésen kívül még az egységbezárás és a polimorfizmus miatt is jelentősen eltér az eljárásorientált nyelvektől. [4][12]

# A Python

A szoftver elkészítéséhez a Python programozási nyelvre esett a választásom. A Python egy rugalmas, egyszerű kódú, dinamikus, objektumorientált és általános célú, magasszintű programozási nyelv, amely széles körben alkalmazható. Ingyenesen használható, nyílt forráskódú, és nagy népszerűségnek örvend, mind a vállalatok, mind a magánszemélyek körében. [17]

Ez a nyelv, különböző programozási stílusokat is támogat, például az objektumorientált illetve a strukturális. Rugalmasságának egyik fő oka, hogy más nyelven írt programokat is képes használni, amiket modulokként importál a saját környezetébe. A beimportált programokat aztán tovább lehet bővíteni, alakítani. [17]

## Története

A Python fejlesztése az 1990-es évek elején indult Hollandiában, ahol Guido Van Rossum a CWI alkalmazottja 1991-ben bemutatta a Python első verzióját. A projekt célja eredetileg az ABC programnyelv helyettesítésére szolgáló nyelv kifejlesztése volt. 1995-ben megjelent a Python 1.2 még szintén a CWI fejlesztésével. [17]

2000-ben elkészült a Python 1.6, majd ezután Guido Van Rossum összeállt a BeOpem.com-al, és megalapították a Python Laboratoriest. Még ugyanabban az évben a Python Laboratories kifejlesztette a Python 2.0-t. A 2.0-ban már olyan újdonságok voltak, mint a listák használata és a keletkezett memóriaszemét számolása. [17]

A 2.0-s verzió kiadása után megalakult a Python Software Foundation, amely a mai napig tulajdonosa és fejlesztője az immár ingyenes és nyílt forráskódú Pythonnak. 2008-ban kiadták a Python 3.0-t, amely ismét jelentős változásokon esett át. Bevezették az integer típusokat, új szintaxisokat hoztak létre, és megváltoztattak régieket, létrejött a print funkció, valamint ez a verzió szándékosan nem kompatibilis a korábbi verziókkal. [17] [22]

## A Python néhány fontos jellemzője:

* gyors és hatékony:
  + Tartalmaz egy standard belső könyvtárat. Minden olyan lehetőséget biztosít, amely a programozáshoz szükséges, az alapműveletektől a speciális funkciókig. Ezek a harmadik féltől származó eszközök mindent lehetővé tesznek a Pythonban. Például egy webszerver létrehozásához elég három sornyi kód megírása. [17]
* más technológiákat is támogat:
  + Többek között támogatja a COM és a .Net objektumokat. Ezen kívül kiegészítőket és alternatívákat is létrehoztak a Python számára, amik megkönnyítik az integrált módban történő munkát az objektumokkal. [17]
* operációs rendszer független:
  + A Python scriptek különböző operációs rendszereken is használhatóak, például Windows, Linux, MacOS. Néhány verzió kiadásra került a .Net, Java és Nokia S60 használatra is. A programok kimenete az összes platformon igencsak hasonló. [17]
* egyszerű:
  + Ez egy nagyon magas szintű nyelv, amely a sok forrásnak köszönhetően könnyen megtanulható. A harmadik féltől származó eszközök sokfélesége miatt a felhasználókat folytonos tanulásra és fejlesztésre ösztönzi. [17]
* nyílt forráskódú:
  + Annak ellenére, hogy minden jog a Python Software Foundation kezében van, nyílt a forráskódja, és semmilyen korlátozáshoz nincs kötve a használata, terjesztése vagy a módosítása. [17]

## További tulajdonságok

A Python egy több paradigmás programnyelv: az objektumorientált programozás és a strukturált programozás teljes mértékben támogatott, és számos olyan nyelvi funkció létezik, amelyek támogatják a funkcionális programozást és a szempontorientált programozást. Sok más paradigma támogatott a kiterjesztések használatával, ideértve többek között a logikai programozást. [17]

A Python a memóriakezelést a dinamikus gépelés, a referenciaszámlálás és a ciklusérzékelő szemétgyűjtés kombinációjával oldja meg. Egy másik fontos tulajdonság a dinamikus névfeloldás, ami a metódus és a változó nevét a program végrehajtása során köti össze. [17]

## Milyen programok írásához jó?

A Python egy jól megtervezett programozási nyelv, amely kiválóan alkalmas a valós világban előforduló problémák programozói megoldására. Az alábbiakban összeszedem a Python nyelven írható leggyakoribb programtípusokat:

* rendszer program
  + Az operációs rendszer szolgáltatásaival való együttműködéshez létrehozott belső Python interfészek a Pythont megfelelő nyelvvé teszik a rendszer programok létrehozásához. Ezek az interfészek bizonyos funkciókat biztosítanak, például: fájlok és könyvtárak műveletei, párhuzamos feldolgozás. A rendszererőforrásokkal való együttműködéshez is tartalmaz néhány eszközt, például: környezeti változók, fájlok, socketek, folyamatok, parancssor, szabványos stream-interfészek, shell programozás. [17]
* grafikus felhasználói felület (GUI)
  + A Tkinter egy szabványos objektumorientált felület, amelyet a Python fordító használ. Ez biztosítja a GUI tervezésének alapvető eszközeit. A generált GUI-k pmw formátumban vannak, és hasonló a megjelenésük a különböző platformokon. Egy másik felület a wxPython, amely a C++ könyvtárra épül. A Tkinter és a wxPython az alapvető interfészek a GUI-k tervezéséhez a Pythonban. Ezen kívül számos magas szintű GUI interfész létezik, amelyek megkönnyítik a munkát. [17]
* hálózati és internetes program
  + Különböző modulok vannak beágyazva a Python szabvány könyvtárába, amelyek számos eszközt kínálnak a hálózati programozók számára, például: kliens-szerver kapcsolat, socket programozás, FTP, Telnet, e-mail funkciók, RPC, SOAP. Néhány harmadik féltől származó eszköz, például a mod-Python, lehetővé teszi az apache-hoz hasonló webszerverek számára a Python-szkriptek futtatását. Továbbá néhány népszerű program, például: Django, Turbo gears, Pylon, Zope és Web Ware támogatja a Python szkripteket. [17]
* adatbázis
  + A legtöbb általános adatbázist, mint például a Sybase, az Oracle, az Informix, a MySQL, a PostgreSQL, az SQLite, támogatják a Python interfészek. Az előre definiált adatbázis API-k segítik a programozókat abban, hogy hasonló kódokon keresztül különböző adatbázisokkal dolgozzanak. A Pickle egy szabványos modul, amellyel fájlokban tárolhatjuk és helyreállíthatjuk az objektumokat. Ezen kívül a ZODB egy tisztán objektumorientált eszköz az adatbázisokkal való munkához. A Python 2.5-től kezdődően az SQLite-t a Python szabványos részének tekintik. [17]
* numerikus program
  + A NumPy modullal rendelkező Python hatékony alternatíva lehet a FORTRAN és a C++ számára, mivel hatékony eszközöket kínál matematikai könyvtárakkal való együttműködéshez, egyszerű Python kódok használatával. Az interneten számos harmadik féltől származó eszköz található a Python által végzett numerikus programozáshoz. [17]
* komponens integritás
  + A Python képes integrált kapcsolatot létesíteni kódjai és más alkatrészek között. Az olyan eszközök, mint a Swing és a SIP, importálhatják más nyelvek fordított kódjait a Pythonban történő használatra. [17]

# Gépi látás

A gépi látás lényege, hogy egy modellt alkossunk a valódi világról képek segítségével. A gépi látás rendszere hasznos információkat gyűjt a képről a kétdimenziós vetületei segítségével. Mivel a képek a háromdimenziós világ kétdimenziós vetületei, ezért az információk nem állnak közvetlenül rendelkezésre, vissza kell állítani azokat. Ez a helyreállítás megköveteli a több az egyhez (many-to-one) leképezés inverzióját. Az információk visszaállításához kell a képen szereplő objektumok ismerete, és vetületgeometria is szükséges. [10]

## Alkalmazási területei

Számos terület kapcsolódik a gépi látáshoz. Mint ebből a részből kiderül, sok területen fejlesztettek ki különböző technikákat a képekből való információk kinyerésére. Ebben a részben röviden bemutatok néhány területet, amelyek szorosan kapcsolódnak a gépi látáshoz.

* Képfeldolgozás
  + A képfeldolgozás egy jól fejlett terület. A képfeldolgozási technikák általában képeket alakítanak át más képekké: az információ-helyreállítás a felhasználó feladata. Ez a terület olyan témákat tartalmaz, mint a képjavítás, a képtömörítés és az elmosódott vagy életlen képek javítása. Másrészt a gépi látás algoritmusai képeket vesznek bemenetként, de más típusú kimeneteket hoznak létre, például ábrázolják az objektum kontúrjait egy képen. A gépi látás hangsúlyt fektet az információk automatikus helyreállítására, minimális emberi interakcióval. Általában bizonyos információk javítására és a zaj elnyomására szolgálnak. [10]
* Grafika
  + A számítógépes grafika olyan geometriai primitívekből állít elő képeket, mint a vonalak, körök és szabad formájú felületek. A számítógépes grafikai technikák jelentős szerepet játszanak a megjelenítésben és a virtuális valóságban. A számítógépes grafika a képek szintézise, a gépi látás pedig a képek elemzése. Kezdetben e két terület között nem volt sok kapcsolat, de az utóbbi években ez a két terület egyre szorosabbá vált. A gépi látás görbe és felületi ábrázolásokat, valamint a számítógépes grafikák számos egyéb technikáját használja, a számítógépes grafika pedig a gépi látástól kezdve számos modell felhasználásával valósítja meg a valósághű képeket. A vizualizáció és a virtuális valóság közelebb hozta ezt a két területet. [10]
* Mintafelismerés
  + A mintafelismerés a numerikus és a szimbolikus adatokat osztályozza. Számos statisztikai és szintaktikai technikát fejlesztettek ki a minták osztályozására. A mintafelismerés technikái fontos szerepet játszanak a gépi látásban és az objektumok felismerésében. Valójában számos ipari alkalmazás nagyban támaszkodik a mintafelismerésre. Az objektumfelismerés általában más technikákat igényel. [10]
* Mesterséges intelligencia
  + A mesterséges intelligencia az intelligens rendszerek tervezésével és az intelligencia számítási aspektusainak tanulmányozásával foglalkozik. A mesterséges intelligenciát a képek elemzésére használják a képek tartalmának szimbolikus ábrázolásával, a jellemzőik megszerzése érdekében. A mesterséges intelligenciának három szakaszát tekinthetjük: észlelés, megismerés és cselekvés. Az észlelés a világból érkező jeleket szimbólummá alakítja, a megismerés manipulálja a szimbólumokat, a cselekvés pedig a szimbólumokat olyan jelekké alakítja, amelyek befolyásolják a világ változását. A mesterséges intelligencia számos technikája fontos szerepet játszik a gépi látás minden aspektusában. Valójában a gépi látást gyakran a mesterséges intelligencia részterületének tekintik. [10]
* Neurális hálózatok
  + A neurális hálózati alapú eszközöket gyakran használják az alakzatok jelenlétének meghatározására, vagy arra, hogy a képen lévő objektum jó vagy rossz. Ezek az eszközök a képosztályozóként ismert algoritmusok csoportjába tartoznak, kezdve a példányalapú osztályozóktól, például a k-legközelebbi szomszédtól (k-NN) a döntési fa osztályozókig. Ezek közül számos felhasználható a gépi látásban. Az olyan vállalatok, mint az MVTec Software, már előre képzett ideghálózatokat, támogató vektorgépeket, Gauss-keverék modelleket és k-NN osztályozókat kínálnak. Meg kell jegyezni, hogy a mély tanulási hálózatok több százezer mintaképet igényelhetnek minden hibakategóriához az érvényes felismerési eredmények eléréséhez. Azonban az átfogó mély tanulási hálózatoknak viszonylag kevés mintaképre van szükség a képzéshez, ami jelentős időt takarít meg. [10]

# A program megtervezése

Mint azt korábban már leírtam, a szakdolgozat célja egy zenei kotta digitalizálását végrehajtó szoftver elkészítése. A projekt sikeres megvalósításához a programozás, a gépi látás és képfeldolgozás különböző eszközeit kell jól kezelni. Ezen három terület ismeretével és megfelelő alkalmazásával lehet létrehozni a kívánt programot. A következő szegmensben bemutatom azokat az eszközöket és technikákat, amelyeket fel fogok használni a projekt megvalósítása során.

A korábbi részekben már kifejtettem, hogy a Python programozási nyelvet fogom használni a program megírásához. A Python programnyelvben belül található egy csomag, amit OpenCV-nek hívnak. Az OpenCV a gépi látás, a gépi tanulás és a képfeldolgozás hatalmas, nyílt forráskódú könyvtára. Használatával képeket és videókat lehet feldolgozni a tárgyak, arcok vagy akár kézírás azonosítására. Különböző könyvtárakkal, például a Numpy-al integrálva a Python képes feldolgozni az OpenCV tömbstruktúráját elemzés céljából. A képminták és különféle jellemzőinek azonosításához vektorokat használ, és matematikai műveleteket hajt végre ezeken a tulajdonságokon. [11][16]

Nagyon sok alkalmazás megoldható az OpenCV használatával, ezek közül néhány:

* arcfelismerés
* forgalomszámlálás
* autók sebességének mérése
* hibák észlelése gyártási folyamatokban
* google utcakép (street view)
* videó/kép keresés és visszaállítás
* sofőr nélküli autó navigáció és vezérlés
* objektum felismerés
* orvosi képelemzés

[11][16]

## Kép előfeldolgozása

A gépi látás rendszerek egyik legfontosabb problémája az objektumokat ábrázoló részképek azonosítása. Ez a művelet, amely annyira természetes és olyan egyszerű az emberek számára, meglepően nehéz a számítógépek számára. A kép régiókra osztását szegmentációnak nevezzük. Ideális esetben a partíció egy objektumot vagy annak egy részét képviseli. Formálisan a szegmentálást úgy határozhatjuk meg, hogy egy képet, az F [i, j] -t részképekre, régiókra osztjuk, oly módon, hogy minden részkép egy lehetséges objektum. [10]

A szegmentálás a pixelek régiókba csoportosítása, oly módon, hogy:

(1)

(2)

Az (1) egyenletben a {Pi} egy kimerítő particionálás.

A (2) egyenletben a {Pi} egy exkluzív particionálás.

Minden Pi régió kielégít egy alapot, vagyis a partició minden pontjának van valamilyen közös tulajdonsága. A szomszédos régiókhoz tartozó pixelek, ha együtt veszik őket, nem felelnek meg az alapnak. [10]

Amint a fentiekből látható, a partició megfelel az alapnak. Ez az alap lehet rendkívül egyszerű is, például egyenletes intenzitású, de a legtöbb alkalmazásban összetettebb. A szegmentálás nagyon fontos lépés a képek megértésében. [10]

A szürkeárnyalatos kép megfelelő szegmentálásával bináris képet kapunk. Ha egy objektum intenzitási értékei intervallumban vannak, és a háttérpontok intenzitási értékei ezen az intervallumon kívül esnek, akkor bináris képet kaphatunk egy küszöbölés művelet segítségével, amely az intervallum pontjait 1-re, a tartományon kívül pedig 0-ra állítja. Így a bináris látás esetében a szegmentálás és a küszöbérték azonos. Sok kamerát terveztek a küszöbölés hardveres végrehajtására. Az ilyen kamera kimenete bináris kép. A legtöbb alkalmazásban azonban a kamerák szürkeárnyalatos képet adnak, és a bináris képet a küszöbérték alkalmazásával nyerik ki. [10]

A küszöbölés egy olyan módszer, amellyel a szürkeárnyalatos képet bináris képpé alakíthatjuk át úgy, hogy az érdeklődésre számot tartó objektumok elkülönüljenek a háttértől. Ahhoz, hogy a küszöbölés hatékony legyen, az objektum és a háttér szétválasztásban, szükséges, hogy az objektumok és a háttér közötti kontraszt kellően nagy legyen, és ismerjük az objektumok vagy a háttér intenzitásának szintjét. Fix küszöbölési sémában ezek az intenzitási jellemzők határozzák meg a küszöbértéket. [10]



1. ábra: A kép bal oldalán a szürkeárnyalatos kép, jobb oldalán a bináris kép látható.

Forrás: https://pippin.gimp.org/image-processing/gen\_images/threshold.png (megtekintve: 2021.04.25.)

Tegyük fel, hogy a B [i, j] bináris kép megegyezik az FT [i, j] bináris szürke képpel, amelyet az eredeti F [i, j] szürke kép T küszöbérték felhasználásával kapunk. Így,

(3)

sötétebb objektum világos háttéren:

(4)

Ha tudjuk, hogy az objektum intenzitási szintje [T1, T2] intervallumba esik, akkor használhatjuk:

(5)

Az általános küszöbölési sémát, amelyben egy objektum intenzitási szintje több diszjunkt intervallumból származhat, ábrázolhatjuk úgy, hogy:

(6)

ahol Z az objektumkomponensek intenzitási értékeinek halmaza. [10]

Vegyük figyelembe, hogy az alkalmazás tartományról szóló tudás hogyan épül be a küszöbölő algoritmusba. Valójában a tartományra szabták; ezért ugyanazok a küszöbértékek nem biztos, hogy működnek egy új tartományban. A küszöbértéket általában az alkalmazás tartományával kapcsolatos tapasztalatok alapján választják ki. Bizonyos esetekben a rendszer első néhány futtatását fel lehet használni egy kép interaktív elemzésére és a küszöbérték megfelelő meghatározására. [10]

A képek automatikus küszöbölése gyakran az első lépés a képek elemzésében a gépi látás rendszerekben. Számos technikát fejlesztettek ki a kép intenzitáseloszlásának és az objektumok felismerésének felhasználására, a megfelelő küszöbérték automatikus kiválasztására. [10]

## Vonalkeresés

A zenei kották meglehetősen sok egyenes vonalat tartalmaznak. Az öt hosszú fő vonalat kottavonalaknak hívják, ezek mindig egyenesek és párhuzamosak. Ezeken kívül az egyes hangjegyek is tartalmaznak vízszintes, illetve függőleges vonalakat is. Ezeket a vonalakat fel kell ismerni és meg kell határozni az elhelyezkedésüket.

### Hough Line Transform

A Standard Hough Transform (továbbiakban SHT) egy körök, vonalak és egyéb egyszerű alakzatok megtalálására alkalmas módszer. Az eredeti Hough Transform csak vonal transzformáció volt, amely viszonylag gyors megoldás egy vonal megtalálására egy bináris képen. [2][5][10]

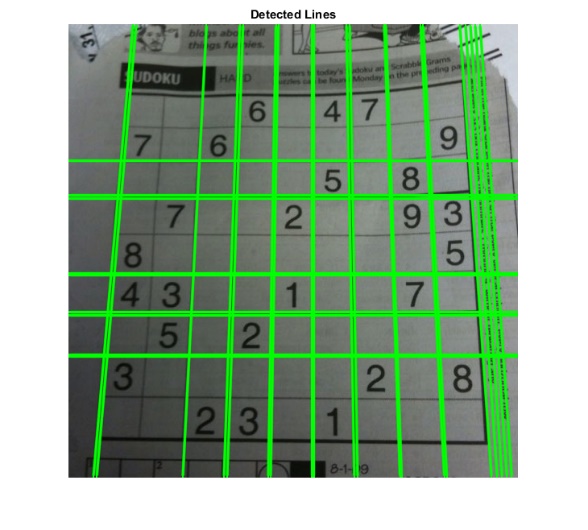
Az SHT alapelmélete az, hogy a bináris kép bármely pontja része lehetne a lehetséges vonalaknak. Ha az egyes vonalakat ’a’ meredekség és ’b’ metszéspont alapján paraméterezzük, akkor az eredeti kép egy pontja átalakul az (a, b) síkbeli pontok azon helyévé, amely megfelel az adott ponton áthaladó összes vonalnak. Ha a bemeneti kép minden pixeljét, ami nem nulla, a kimeneti kép ilyen pontjainak halmazává konvertáljuk, és összesítjük az összes ilyen átalakítást, akkor a bemeneti képen megjelenő vonalak helyi maximumokként jelennek meg a kimeneti képen. Mivel az egyes pontokból származó átalakításokat összegezzük, az (a, b) síkot általában felhalmozó síknak nevezzük. [2][5][10]

Előfordulhat, hogy a meredekség és metszéspont módszere nem igazán a legjobb módja annak, hogy az összes pontot áthaladó vonalat ábrázolja, mivel a meredekség függvényében a vonalak jelentősen eltérő gyakoriságúak, valamint a lehetséges meredekségek intervalluma végtelen lehet. Éppen ezért némileg eltér a numerikus összeállításban használt transzformációs kép tényleges paraméterezése. Az előnyben részesített paraméterezés mindegyik egyenest poláris koordinátákban (p, 0) jeleníti meg, az implicit vonal a jelzett ponton áthaladó, de az origótól az adott pontig sugárirányra merőleges vonal. Egy ilyen egyenes egyenlete:

(7)

Az (x, y) az egyenes vonalon lévő pontok koordinátáit, vagyis a pixelteret jelöli, míg P és 0 a paramétertér. Az összes (P, ()) paraméter esetében az algoritmus meghatározza, hogy van-e elegendő bizonyíték egy valódi vonalra az adott pixelnél, amely kiszámítja a paraméterértékeket és felhalmozza a tárolókat a paramétertérben az összes képpont alapján. [2][5][10]

A Hough transzformációs számítás három részből áll: (1) a paraméterértékek kiszámítása és a tárolók felhalmozása a paramétertérben; (2) a vonalszakaszokat ábrázoló lokális maximumok megkeresése; és (3) a vonalszakaszok kivonása az ismeretek felhasználásával a maximális pozíciókból. [2][5][10]



2. ábra: A Standard Hough Line Transform által felismert vonalak egy Sudokun

Forrás: http://amroamroamro.github.io/mexopencv/opencv/hough\_lines\_demo\_02.png (megtekintve: 2021.04.26.)

Az algoritmus egyszer végigmegy a kép minden pixelén. Ezért időbeli bonyolultsága lineáris a teljes képszámmal, amely a kép szélességének és magasságának szorzata. Ugyanakkor, mivel az algoritmusnak fel kell halmoznia az egyes pixelek értékeit, tárolási igénye a pixelszámmal együtt változik. Tehát az algoritmus számítási komplexitása és tárolási követelményei jelentik az SHT fő szűk keresztmetszeteit. Ezért az SHT számos változatát javasolták a számítási és tárolási terhek enyhítésére. [2][5][10]

Az OpenCV Hough transzformációs algoritmus nem teszi egyértelművé ezt a számítást a felhasználó számára. Ehelyett egyszerűen visszaadja a helyi maximumokat a (p, 0) síkban. Azonban a vonaltranszformációs függvény argumentumainak megértéséhez azonban ezt a folyamatot meg kell értenünk. [2][5][10]

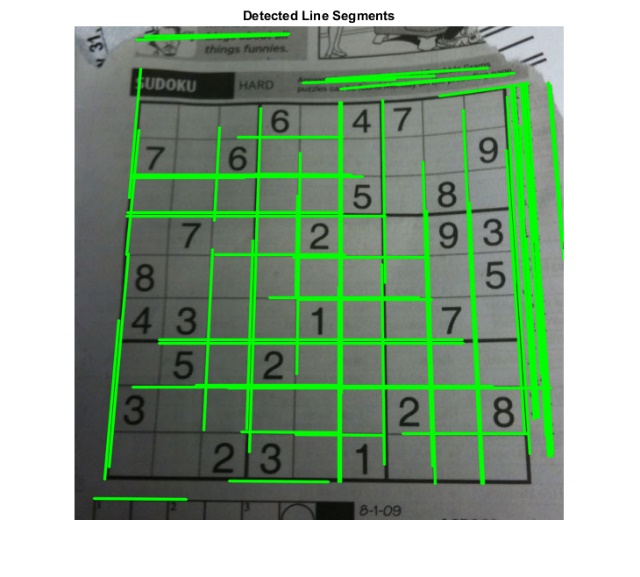
### Probabilistic Hough Transform

A Probabilistic Hough Transform (továbbiakban PHT) a Hough Line Transform egyik variációja, amely többek között az elhelyezkedése mellett, a méretét is kiszámítja a vonalaknak. Azért ’probabilistic’ (valószínűségi), mert ahelyett, hogy felhalmozná az összes lehetséges pontot az felhalmozás síkjában, csak a töredéküket halmozza fel. Az elképzelés az, hogy ha a keresett dolog mindenképp elég nagy lesz, akkor az adott idő töredéke is elég lesz, hogy megtaláljuk; ennek a valószínűségi módszernek az eredménye jelentősen csökkentheti a számítási időt. Mindkét algoritmus ugyanazzal az OpenCV függvénnyel érhető el, bár néhány argumentum jelentése eltérő lehet attól függően, hogy melyik módszert használjuk. [5][7]

Az első argumentum a bemeneti kép. 8 bites képnek kell lennie, de a bemenetet bináris információként kezeljük. A második argumentum egy olyan hely mutatója, ahol az eredmények tárolhatók, amely lehet memóriatároló vagy sima [N,1] mátrix tömb. A következő argumentum a metódus, ami lehet CV\_HOUGH\_STANDARD, CV\_HOUGH\_PROBABILISTIC vagy CV\_HOUGH\_MULTI\_SCALE a PHT-hoz, vagy a SHT többskálás változatához. [5][7]

A következő két argumentum az rho és theta. Ezek adják meg a vonalak kívánt felbontását (a felhalmozó sík felbontását). Az rho egységei a pixelek, a téta egységei a radiánok; így az felhalmozó sík egy kétdimenziós hisztogramnak tekinthető, ahol a két dimeznió cellái az rho pixelei és theta radiánjai. A küszöbérték a felhalmozó síkban az az érték, amelyet el kell érni ahhoz, hogy a rutin felismerje a vonalat. Ez az utolsó argumentum a gyakorlatban kissé trükkös, ugyanis nincs normalizálva, ezért számíthatunk arra, hogy a kép méretét fel kell nagyítanunk. Azt se feledjük, hogy ez az argumentum valójában azon a pontok számát is megadja, amelyeknek az adott vonalon kell lennie, hogy a rutin felismerje a vonalat. [5][7]

A param1 és param2 argumentumokat az SHT nem használja. A PHT esetében a param1 meghatározza a felismerendő vonalszakasz minimális hosszát, a param2 pedig az egy egyenesre eső szegmensek közötti elválasztást, amely szükséges ahhoz, hogy az algoritmus ne egyesítse őket egyetlen hosszabb szakaszba. A többskálás HT esetében a két paraméter arra szolgál, hogy nagyobb felbontást érjünk el, amelyekre a vonalak paramétereit ki kell számítani. A többskálás HT először kiszámítja a vonalak helyét az rho és a theta paraméterek által megadott pontossággal, majd tovább finomítja ezeket az eredményeket param1 és param2 tényezőkkel. [5][7]



3. ábra: A Probabilistic Hough Transform által felismert vonalak egy Sudokun

Forrás: http://amroamroamro.github.io/mexopencv/opencv/hough\_lines\_demo\_03.png (megtekintve: 2021.04.26.)

A függvény eredménye attól függ, hogyan hívták meg. Ha a line\_storage értéke egy mátrix tömb volt, akkor a tényleges visszatérési érték NULL lesz. Ebben az esetben a mátrixnak CV\_32FC2 típusúnak kell lennie, ha az SHT-t vagy a többskálás HT-t használja, és CV\_32SC4-nek kell lennie, ha a PHT-t használja. Az első két esetben az egyes sorok p és 0 értékei a tömb két csatornájába kerülnek. A PHT esetében a négy csatorna megtartja a felismert szegmensek kezdő és végpontjának x és y értékeit. Mindegyik esetben a tömb sorainak számát a cvHoughLines2 () frissíti, hogy megfelelően tükrözze a megtalált vonalak számát. [5][7]

Ha a line\_storage érték egy memória tároló mutatója volt, akkor a visszatérési érték egy CvSeq szekvencia struktúra mutatója lesz. Ebben az esetben az egyes vonalakat vagy vonalszegmenseket az alábbi paranccsal kapjuk meg:

float\* line = (float\*) cvGetSeqElem(lines , i);

ahol a lines a cvHoughLines2() visszatérési értéke, az i pedig a keresett vonal indexe. Ebben az esetben a vonal egy mutató lesz az adott vonal adataihoz, a line[0] és a line[1] pedig a lebegőpontos p és 0 értékek lesznek, vagy a CvPoint struktúrák lesznek a szegmensek végpontjaihoz. [5][7]

### Morfológiai műveletek

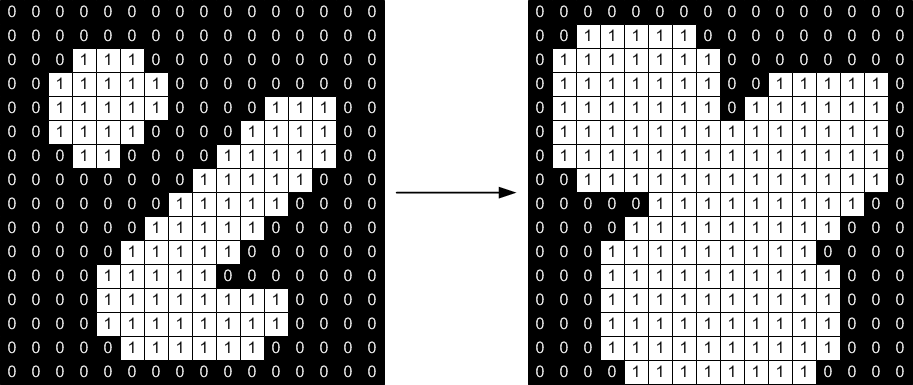
A morfológia alaktant jelent, és egyaránt alkalmazzák a biológiában és a matematikában is. A matematikában halmazokon elvégezhető műveleteket jelent, ezt használja a képfeldolgozáshoz is. A morfológia célja, hogy a struktúrák és formák tanulmányozása révén valamilyen releváns adatot nyerjünk ki a képből. Ezek a műveletek gyorsan implementálhatók és széles körben alkalmazhatók. Az alapműveletek akkor a leghatékonyabbak, ha bináris képen alkalmazzák őket, ezért én is így tettem.

A feldolgozandó kép mellett azonban szükség van egy strukturáló elemre is, amely méretében jóval kisebb a vizsgált képnél, mivel ez fogja végigjárni a kép teljes területét. Ennek a strukturáló elemnek (más néven kernelnek) három különböző alakja lehet. A kernel formája lehet ellipszis, téglalap vagy kereszt. [9]

Az 1-esek jelölik azt a területet, ahol az adott művelet hatása alkalmazva lesz az adott pixelen, a 0-val jelölt területeket pedig érintetlenül hagyja.

A morfológiai szűrők vagy morfológiai műveletek arra használhatóak fel, hogy a kép egyes területeit felnagyítsuk vagy esetleg kicsinyítsük, illetve, hogy eltávolítsuk vagy kitöltsük egy terület körüli képpontokat. Az alábbiakban ismertetem ezeket az alapvető műveleteket, amelyek a bemeneti képen lévő objektumokat a kiválasztott strukturáló elemben kódolt jellemzők alapján dolgozzák fel.

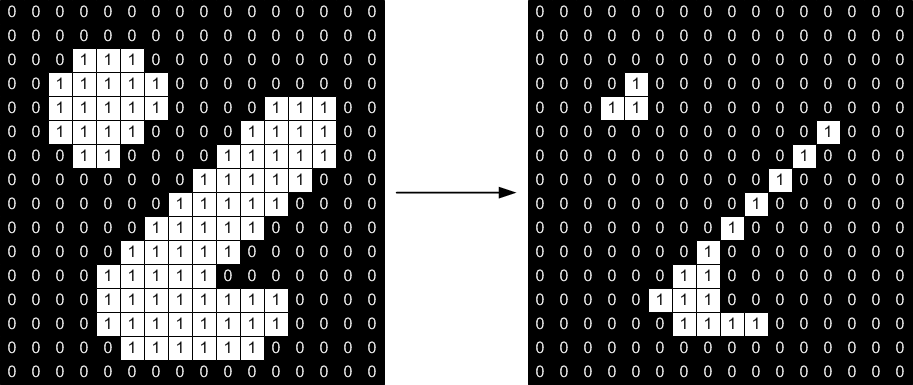
A tágulás (dilation) egyike a két morfológiai alapműveletnek. Ez az operátor azt a hatást éri el, hogy a kép előterében lévő pixelek körüli határokat folyamatosan növeli. Ezzel az előtérben lévő pixelek mérete növekszik, a közöttük lévő lyukak pedig csökkennek. A tágulás művelet két adatot vesz be bemenetként. Az első a kép, amelyen a műveletet végre kívánjuk hajtani. A második azon koordinátapontok halmaza, amelyet strukturáló elemként ismerünk. Ez a strukturáló elem határozza meg a tágulás pontos hatását a bemeneti képre. A bemeneti kép pixeljeit a bal felső saroktól kezdve megvizsgáljuk a strukturáló elemmel. Minden háttérpixelnél (amit bemeneti pixelnek nevezünk) a strukturáló elemet a bemeneti kép tetejére helyezzük úgy, hogy a strukturáló elem origója egybeessen a bemeneti pixel pozíciójával. Ha a strukturáló elemben legalább egy pixel egybeesik az alatta lévő kép előtérbeli pixelével, akkor a bemeneti pixel az előtér értékére lesz állítva. Ha azonban a képen az összes megfelelő pixel háttérben van, a bemeneti pixel a háttér értékén marad.[9]



4. ábra: Példa 3x3-as téglalap strukturáló elemmel végzett tágulásra

Forrás: <http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05_Image%20Processing/Morphology%20Filters.htm> (megtekintve: 2021.11.13.)

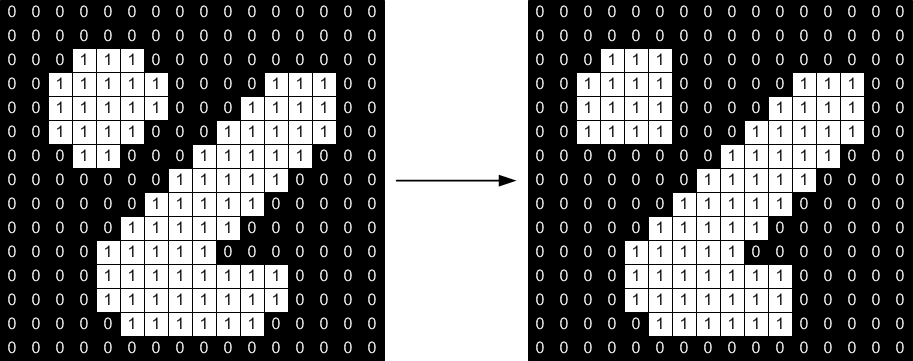
Az erózió a két morfológiai alapművelet közül a másik. Ez az operátor azt a hatást éri el, hogy a kép előterében lévő pixelek körüli határokat folyamatosan csökkenti. Így az előtérben lévő pixelterületek mérete csökken, a lyukak pedig nagyobbakká válnak. A táguláshoz hasonlóan az erózió is két adatot vár bemenetként. Az első a bemeneti kép, a második pedig a strukturáló elem. A művelet végrehajtása is megegyezik az előzőével, azaz a strukturáló elem segítségével a bemeneti kép minden egyes pixeljét megvizsgáljuk, és amennyiben a strukturáló elem minden pixeléhez az alatta lévő kép megfelelő pixelje előtérben lévő pixel, akkor a bemeneti pixel marad úgy, ahogy van. Ha azonban a képen a megfelelő pixelek bármelyike háttér, akkor a bemeneti pixel is háttérértékre lesz állítva.[9]



5. ábra: Példa 3x3-s strukturáló elemmel végzett erózióra

Forrás: [http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05\_Image%20Processing/Morphology%20Filters.htm (megtekintve](http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05_Image%20Processing/Morphology%20Filters.htm%20%20(megtekintve): 2021.11.13)

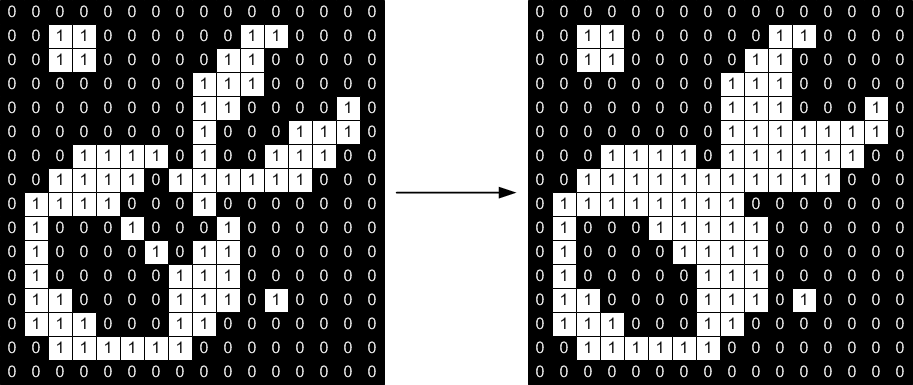
A nyitás egy az erózióból származtatott operátor. Szintúgy bináris képeken alkalmazzák főként, de akár szürkeárnyalatos képen is működőképes. Az algoritmus hatása az erózióhoz hasonló, ugyanis ennek is az a feladata, hogy az egyes pixelterületek széleit eltűnteti. Azonban ez általánosságban kevésbé ’pusztító’, mint az erózió. Más morfológiai műveletekhez hasonlóan a pontos működést itt is a strukturáló elem határozza meg. Az művelet célja az, hogy megőrizze azokat az előtér-régiókat, amelyek hasonló alakúak ehhez a strukturáló elemhez, vagy amelyek teljesen tartalmazhatják a strukturáló elemet, miközben kiiktatja az előtér pixeleinek összes többi régióját. Nagyon leegyszerűsítve a nyitást úgy definiálhatjuk, mint egy eróziót, amelyet egy tágulás követ, mindkét művelethez ugyanazt a strukturáló elemet használva. Míg az erózió elég hatékonyan felhasználható a nemkívánatos előtérbeli pixelek kis csomóinak kiküszöbölésére, pl. 'sózaj', nagy hátránya, hogy az előtérben lévő pixelek minden régióját válogatás nélkül érinti. A nyitás ezt úgy tudja megkerülni, hogy az eróziót és a tágulást is végrehajta a képen. A bemeneti képen elindítja a strukturáló elemet, és csúsztatja körbe az egyes előtér-területeken belül anélkül, hogy megváltoztatná a tájolását. Az összes képpont, amelyet a strukturáló elem lefedhet, és a strukturáló elem teljes egészében az előtérben van, megmarad. Azonban minden olyan előtér pixel, amelyet a strukturáló elem nem tud elérni anélkül, hogy egyes részei kimozdulnának az előtérből, erodálódnak. A nyitás elvégzése után az előtérrégiók új határai mind olyanok lesznek, hogy a strukturáló elem beleférjen, így az azonos strukturáló elemmel rendelkező további nyitásoknak nincs hatása.[9]



6. ábra: Példa 3x3-as strukturáló elemmel végzett nyitásra

Forrás: [http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05\_Image%20Processing/Morphology%20Filters.htm (megtekintve](http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05_Image%20Processing/Morphology%20Filters.htm%20%20(megtekintve): 2021.11.13)

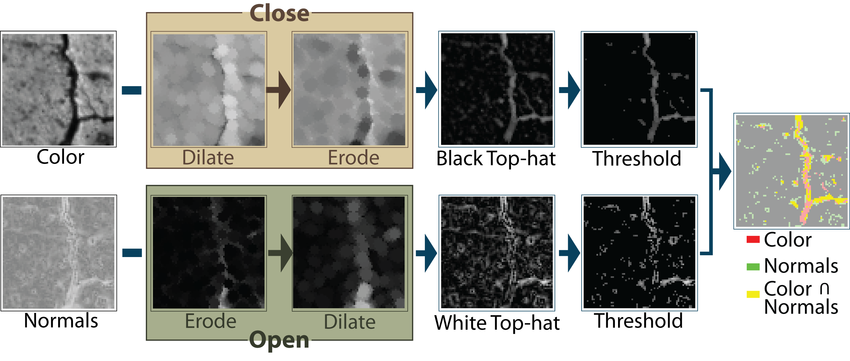
A zárás az nyitáshoz hasonlóan az erózió és a tágulás alapvető műveleteiből származtatható. Az említett operátorokhoz hasonlóan általában bináris képekre alkalmazzák, bár vannak szürkeárnyalatos verziók is. A zárás bizonyos tekintetben hasonlít a táguláshoz, mivel hajlamos megnagyobbítani a kép előtér régióinak határait és csökkenteni a lyukakat az ilyen területeken, de kevésbé roncsolja az eredeti határalakot. A többi művelethez hasonlóan a pontos működést egy strukturáló elem határozza meg. Az operátor hatása az, hogy megőrizze azokat a háttérrégiókat, amelyek hasonló alakúak, mint a strukturáló elem, vagy amelyek teljes mértékben tartalmazzák a strukturáló elemet, miközben kiiktatja a háttér pixelek összes többi régióját. A zárás tulajdonképpen egy nyitás, csak fordított sorrendben. Egyszerűen úgy definiálható, mint egy tágulás, amelyet erózió követ, mindkét művelethez ugyanazt a strukturáló elemet használva. A tágulás egyik felhasználási módja a képek háttérpixelein lévő lyukainak kitöltése, pl. 'paprika zaj'. Ezzel azonban az egyik probléma az, hogy a tágulás a pixelek minden régióját válogatás nélkül megváltoztatja. Azzal, hogy a tágulás után a képen eróziót végzünk, ennek a hatásnak egy részét csökkentjük. A kép tetejétől elindul a strukturáló elem az egyes előtér-területeken, és végigcsúsztatja anélkül, hogy megváltoztatná a tájolását. Bármely háttér-határpont esetében, ha a strukturáló elem hozzáérhet a ponthoz anélkül, hogy az elem bármely része egy előtér-területen belül lenne, akkor az a pont háttér marad. Ha ez nem lehetséges, akkor a pixel előtérbe kerül. A zárás elvégzése után a háttérrégió olyan lesz, hogy a strukturáló elem a háttér bármely pontját lefedheti anélkül, hogy annak bármely része előtérpontot is takarna, így a további zárásoknak nincs hatása.[9]



7. ábra: Példa 3x3-as strukturáló elemmel végzett zárásra

Forrás: [http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05\_Image%20Processing/Morphology%20Filters.htm (megtekintve](http://www.theobjects.com/dragonfly/dfhelp/4-0/Content/05_Image%20Processing/Morphology%20Filters.htm%20%20(megtekintve): 2021.11.13)

A matematikai morfológiában és a digitális képfeldolgozásban a cilindertranszformáció egy olyan művelet, amely kis elemeket és részleteket von ki adott képekből. Kétféle cilindertranszformáció létezik – a fehér cilindertranszformáció és a fekete cilindertranszformáció. A fehér cilinder transzformáció a bemeneti kép és annak valamilyen strukturáló elem általi nyitása közötti különbségként, míg a fekete cilinder transzformáció a zárás és a bemeneti kép különbségeként definiálható. A cilinder átalakítások különféle képfeldolgozási feladatokhoz használhatók, mint például a jellemzők kivonása, a háttér kiegyenlítése, a képjavítás és egyebek. Figyelembe kell venni, hogy a transzformációkkal kinyert elemek mérete vagy szélessége a strukturáló elem kiválasztásával szabályozható. Minél nagyobb a kernel, annál nagyobbak a kivont elemek.[9]



8. ábra: Fekete cilinder és fehér cilinder transzformáció szemléltetése

Forrás: <https://www.researchgate.net/profile/Corey-Toler-Franklin/publication/220183814/figure/fig2/AS:393990282203145@1470946012049/Erosion-Detection-The-black-top-hat-transform-is-generated-by-applying-a-closing.png> (megtekintve: 2021.11.14)

## Kontúrkeresés

A kontúrokat egyszerűen úgy lehet megmagyarázni, hogy egy objektum határai mentén az összes azonos színű vagy intenzitású folytonos pontot összekötő görbe. A kontúrok hasznos eszközök az alakzatok elemzéséhez, valamint az objektumok felismeréséhez. A pontosabb eredmény érdekében bináris képeket kell használni, tehát mielőtt kontúrokat keresnénk, mindenképpen küszöbértéket kell alkalnaznunk, ha eddig nem történt volna meg. Az OpenCV-ben a kontúrok megkeresése során, fehér tárgyat keresünk fekete háttéren. Tehát a keresett objektumnak fehérnek, a háttérnek pedig fekete színűnek kell lennie. [15][16]

Az OpenCV-ben a findContours() függvényt alkalmazhatjuk a kontúrok megkeresésére. Három argumentum van a cv2.findContours() függvényben, az első az forrás kép, a második a kontúr visszakeresési módja, a harmadik a kontúr közelítési módszere. És ez adja ki a képet, a kontúrokat és a hierarchiát. A CONTOURS a kép összes kontúrjának listája. Minden egyes kontúr egy Numpy tömb, amiben (x, y) az objektum határpontjainak koordinátái. [15][16]

Általában a cv2.findContours() függvényt használjuk a képen lévő objektumok felderítésére. Néha az objektumok különböző helyeken vannak, de bizonyos esetekben egyes alakzatok más alakzatokon belül vannak. Ebben az esetben a külsőt szülőnek, a belsőt pedig gyermeknek nevezzük. Így a kép kontúrjai valamilyen kapcsolatban állnak egymással. És megadhatjuk, hogy az egyik kontúr hogyan kapcsolódik a másikhoz, például, hogy valamilyen más kontúr gyermeke-e, vagy szülője-e. Ennek a kapcsolatnak az ábrázolását Hierarchiának nevezzük. [15][16]

A visszakeresési módoknak több fajtája is van, ezek a következőek:

* cv2.RETR\_LIST
  + Ez a négy mód közül a legegyszerűbb. Egyszerűen lekéri az összes körvonalat, de nem hoz létre szülő-gyermek kapcsolatot. A szülők és a gyerekek ebben a szabályban egyenlőek, és csak sima kontúrok, azaz mindegyik azonos hierarchia szinthez tartozik. [15][16]
* cv2.RETR\_TREE
  + Ez az módszer lekéri az összes körvonalat, és létrehoz egy teljes családi hierarchia listát. Még azt is megmondja, ki a nagypapa, apa, fiú, unoka és még azon túl is. [15][16]
* cv2.RETR\_CCOMP
  + Ez a mód lekéri az összes kontúrt és 2-es szinttű hierarchiába rendezi őket, vagyis az objektum külső kontúrjai az 1-es hierarchiába kerülnek. Az objektum belsejében lévő lyukak vagy egyéb alakzatok kontúrjai (ha vannak) a 2-es hierarchiába kerülnek. [15][16]
* cv2.RETR\_EXTERNAL
  + Ha ezt a módot használjuk, akkor csak a legkülső kontúrokat adja vissza. Minden gyermek kontúr elmarad. [15][16]

A cv2.findContours() harmadik argumentuma a közelítési mód. Fentebb említettem, hogy a kontúrok az azonos intenzitású vagy színű alakok határai. Az alakzat határának (x, y) koordinátáit tárolja. De tárolja-e az összes koordinátát? Ezt a kontúr közelítési módszere határozza meg. [15][16]

Két különböző mód létezik, az első a cv2.CHAIN\_APPROX\_NONE, amely a megtalált elem összes határpontját eltárolja. Ez esetenként azonban felesleges, mivel ha egy vonalat keresünk akkor nincs szükség az összes pontjának eltárolására. Elég számunkra, ha a végpontjait meghatározzuk. Ezt a cv2.CHAIN\_APPROX\_SIMPLE módszerrel tudjuk megvalósítani, amely eltávolítja az összes felesleges pontot és tömöríti a kontúrt, ezáltal memóriát takarít meg. [15][16]

A kontúrok megrajzolásához a cv2.drawContours() függvényt használjuk. Bármilyen alakzat megrajzolására használható, feltéve, hogy megvannak a határpontjai. Első argumentuma a forráskép, a második argumentuma a kontúrok, amelyeket listaként kell neki átadni, a harmadik argumentum a kontúrindex, a többi argumentum pedig a szín, a vastagság és egyéb állítható paraméterek. [16]



9. ábra: Példa a kontúrok keresésére és megrajzolására OpenCV használatával

Forrás: https://robottini.altervista.org/wp-content/uploads/2012/05/contours.png (megtekintve: 2021.04.27.)

## Konvolúciós neurális hálózat

Az objektum felismerés célja a célobjektumok detektálása a képfeldolgozás és a mintafelismerés elemeivel és módszereivel, valamint az objektumok szemantikai kategóriáinak meghatározása és a célobjektum konkrét helyzetének megjelölése a képen. Gyakorlati alkalmazásban nagy kihívást jelent a számítógépes technológia használata az objektumok automatikus felismerésére. A komplex háttér, a zavaró zajok, az alacsony felbontás, a méretek és a pozíciók változásai is mind komolyan befolyásolják az objektumok észlelését. A hagyományos módszer egy kézzel készített funkción alapul, amely egyes nehézségeket nem kezel jól, és általános használatra sem a legalkalmasabb. Emiatt számos módszert dolgoztak ki az objektum észlelés teljesítményének javítására. [6][8][21]

A kovolúciós neurális hálózat (Convolutional Neural Network) egy olyan mély tanulási modell, amely képes a hierarchikus tanulásra, valamint kutatások bizonyítják, hogy az általa használt funkciók jobban alkalmazhatóak az objektumokok szegmentálására és felismerésére, mint a kézzel készített verzió. [6][8][21]

A gépi látás területén a jellemzők kinyerése és osztályozása mindig is nagyon fontos kutatási irány volt. A hagyományos képfeldolgozási feladatokban a kinyert jellemzők gyakran előre megtervezett jellemzők, statisztikai törvényszerűségek vagy előzetes ismeretek alapján történtek. Tehát nem képes az eredeti kép információit átfogóan és pontosan ábrázolni. [6][8][21]

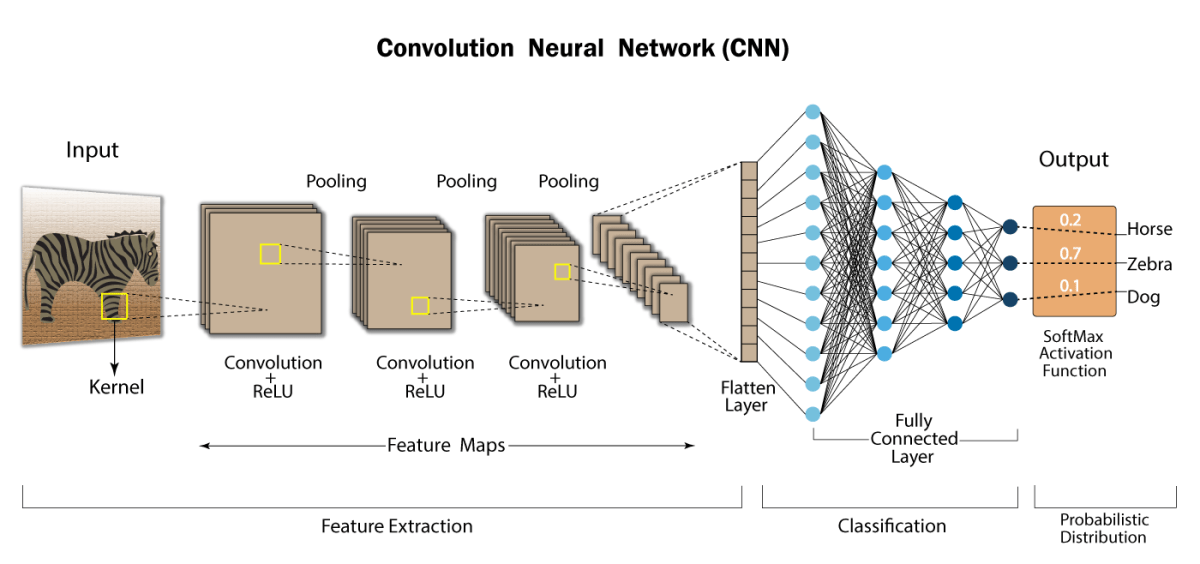
A Convolutional Neural Network (továbbiakban CNN) egy végpontok közötti tanulási modellt nyújt, amelyben a paraméterek a gradiens ereszkedési módszerrel oktathatók. A jól képzett CNN teljesebben elsajátíthatja a kép jellemzőit, és úgy tekinthetünk rá, mint egy fejlettebb fekete dobozra, amely ki is bontja a jellemzőket. [6][8][21]

Az ideghálózat fontos ágaként a CNN növeli a befogadó mező és a megosztott súly fogalmát, ami nemcsak nagymértékben csökkenti a tanítás paramétereit, hanem csökkenti a hálózati modell komplexitását is. Az egyes rétegek jellemzői az előző réteg helyi területéből (befogadó mezőből) származnak a konvolúciós kernel súlyának megosztásával. Ezek a jellemzők teszik a CNN-t alkalmassá a kép jellemzőinek megtanulására és reprezentálására. [6][8][21]

Egy tipikus CNN-ben az első néhány réteg általában váltakozó konvolúciós és összevonási réteg, és a kimeneti réteg közelében lévő hálózat utolsó rétegei általában egy teljesen összekapcsolt hálózat. A CNN oktatása főként az előterjesztő (Forward Propagation) és a visszaterjesztő (Back Propagation) algoritmust használja a rétegkapcsolati súlyok, torzítás és egyéb paraméterek megismerésére. Az oktatás egy felügyelt tanulási folyamat, amely megköveteli a képadatok bevitelét és a megfelelő címkéket a hálózati paraméterek optimalizálásához, melyek alapján végül egy optimalizált súlyú modellt kapunk. [6][8][21]

A CNN különböző funkcionális rétegstruktúrából áll. A tipikus CNN rendelkezik konvolúciós réteggel, összevonási réteggel és teljesen összekapcsolt réteggel. A CNN azonban számos új réteget adhat hozzá az evolúció és a fejlesztés folyamatához, például az SPP (Spatial Pyramid Pooling) réteget, a ROI (Region of Interest) egyesítő rétegét és a R-CNN (Region Based Convolusional Neural Network) réteget. A hagyományos CNN szerkezet javításával jobb teljesítmény érhető el a specifikus problémák megoldására. [6][8][21]

### A hagyományos CNN szerkezete



10. ábra: A hagyományos CNN rétegei

Forrás: https://i0.wp.com/developersbreach.com/wp-content/uploads/2020/08/cnn\_banner.png?fit=1200%2C564&ssl=1 (megtekintve: 2021.04.27.)

Amint az ábrán látható, a tipikus CNN struktúra főként a bemeneti rétegből, a konvolúciós rétegből, az összevonási rétegből, a teljesen összekapcsolt rétegből és a kimeneti rétegből áll. A konvolúciós ideghálózat bemenete általában az eredeti kép. az l-edik réteg j-edik jellemzőtérképét jelöli. Feltételezzük, hogy egy jellemzőtérkép a konvolúciós rétegen, és az alábbiak szerint generálódik:

(8)

A az l réteg funkció térképének megfelelő súlymátrix, a \* a konvolúció szimbóluma, amely specifikus szűrőket ad az l-1 funkció térképeihez, majd a számított eredményeket hozzáadjuk a eltéréshez, végül az –t az f(x) függvény nem lineáris aktiváló műveleteivel kapjuk meg. [21]

Az összevonási réteg, más néven lemásoló réteg, általában a konvolúciós réteget követi, és lemásolja az előző jellemzőtérképet adott szabályok szerint. Ezek a szabályok a következőek: maximum összevonás, átlagos összevonás, sztochasztikus összevonás és átfedő összevonás. Az összevonási rétegnek két fő aspektusa van: a funkció térképek dimenzióinak csökkentése, és a skálák változatlanul tartása. Feltételezzük, hogy egy jellemzőtérkép az összevonási rétegben, és az összevonási műveletet az alábbi formula írja le:

(9)

A pooling(x) függvény a lemásolási szabályt jelöli, pedig az összevonás súlya. Általában egy fix érték, a b eltérést és az f(x) aktiváló műveletet pedig nem használjuk. Tehát az összevonási műveletek általános formája:

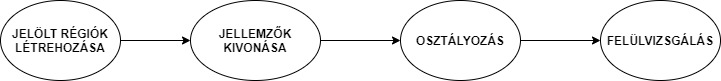
(10)

Teljesen összekapcsolt hálózatban a képek jellemzőtérképeit egydimenziós jellemzővektorokká fűzzük, majd bemeneteként odaadjuk a teljesen összekapcsolt hálózatnak. A teljesen összekapcsolt réteg kimenetét úgy kapjuk meg, hogy a bemenethez súlyozott összegzéseket adunk, majd az aktiváló funkciót alkalmazzuk az alábbi módon:

(11)

[21]

### Objektum felismerés a CNN használatával



11. ábra: A tradícionális objektum felismerés lépései (saját szerkesztés)

Amint az ábrán látható, a hagyományos objektumdetektáló rendszer általában négy szakaszra oszlik. Először is, az adott képen jelölt régiókat hozunk létre csúszóablakos megoldással, majd a releváns jellemzőket kivonjuk ezekből a régiókból. Ezután a betanított osztályozóval a régiók osztályozására és azonosítására kerül sor, végül az eredmények felülvizsgálata és optimalizálása történik meg. [21]

* A jelölt régiók létrehozása

Ez a szakasz az objektum helyének megszerzésére szolgál. Az objektum azonban bárhol lehet a képen. Az objektum mérete és helyzete is bizonytalan, ezért különböző méret- és képarányú csúszóablakokkal végigjárjuk az egész képet. Ez a stratégia szinte az összes helyet magába foglalja, ahol az objektum elhelyezkedhet, de hátrányai is vannak, például az időigény és a redundáns ablakok. Ez súlyosan befolyásolhatja a jellemzők kivonásának gyorsaságát és teljesítményét. [21]

* Jellemzők kivonása

A jellemzők kivonása közvetlenül befolyásolja az osztályozó tervezését és teljesítményét. Nehéz azonban megfelelő funkciót megtervezni a különféle külső tényezők, például a tárgyak mozgása, a megvilágítás változása, az összetett és megváltoztatható környezet miatt. Néhány kézzel készített funkciót széles körben használnak ebben a szakaszban, mint például a skála változatlan jellemzőinek átalakítása (Scale Invariant Feature Transform), az orientált színátmenetek hisztogramjai (Histograms of Oriented Gragients) és a helyi bináris minták (Local Binary Patterns). [21]

* Osztályozás

Az SVM vagy az AdaBoost osztályozókat széles körben használják a kivont jellemzők osztályozására ebben a szakaszban. [21]

* Felülvizsgálás

A besorolás után még mindig sok redundáns ablak van, ezért el kell távolítani a redundáns ablakokat, és optimalizálni kell az észlelési eredményeket a NMS (Non-Maximum Suppression) és az átfedésben lévő határoló doboz (Bounding Box) kombinálásával. A hagyományos objektumészlelésnek két fő problémája van: egyrészt a csúszó ablakon alapuló módszer nem elég céltudatos, nagy időbonyolultságú, és túl sok redundáns ablak van; másodszor, a kézzel készített funkciók nem elég robusztusak a változások sokféleségéhez. [21]

## Régiókon alapuló konvolúciós neurális hálózat

A redundáns ablakok problémájának megoldására a régiójavaslat jó megoldást kínál. Előre kitalálja a képen található obejktumok lehetséges helyét. A régiójavaslat jól használja ki a kép információit, például a színt, az éleket és a textúrát, ami biztosíthatja, hogy a rendszer viszonylag nagy visszahívási lehetőséget kapjon kevesebb ablak esetén. Ez nagymértékben csökkenti a következő műveletek időigényét, és a régiójavaslatok minősége magasabb, mint a csúszóablakoké. 2014-ben Ross B. Girshick javaslata alapján a CNN hagyományos csúszóablakos megoldását leváltották a régiójavaslatos CNN objektum felismerési rendszerére. Ez nagy sikert aratott az objektum detektálásban, és megnyitotta a kapukat a hatékonyabb CNN alapú objektum kereséshez. [21]

A Region Based Convolution Neural Network (továbbiakban R-CNN) három modullal rendelkezik: először létrehoz kategóriafüggetlen régiójavaslatokat. A második modul egy nagy CNN, amely kivonja az adott hosszúságú jellemzővektorokat a régiókból. Az utolsó modul szétválasztja az objektumot és a hátteret lineáris SVM (Support Vector Machine) segítségével. Annak érdekében, hogy növeljük a helymeghatározás pontosságát egy lineáris regressziós modellt képeztek ki a detektálódobozok koordinátáinak módosítására. [18]

Azonban ennek a módszernek is megvannak a hátrányai. Az R-CNN túl sok jelölt ablakot generál, és mivel a CNN-en keresztül vonja ki a jellemzőket, ennek az időigénye még mindig túl nagy. A másik probléma az, hogy a CNN-ek egy fix méretű képet kapnak bemenetként, amelyet az R-CNN vágással vagy egyéb művelettel tud a saját méretéhez illeszteni, ami a képarány és a méretarány módosulását is okozza. [21]

He Kaiming kidolgozta az SSP-Net módszert az R-CNN hibáinak kiküszöbölésére. Az SPP-Net csak egyszer futtatja le a konvolúciós réteget a teljes képen, hogy megkapja a jellemzőtérképeket. A számítási feladatok megosztása miatt a lefutási idő 24-64-szer gyorsabb, mint a sima R-CNN-nél. A javasolt régiók méretei eltérőek, de a teljesen összekapcsolt réteg egy fix méretet vár bemenetként. Ennek megoldására jött létre az SPP réteg. Az SPP réteg az utolsó konvolúciós réteg mögé került, a jelölt ablakok szelektív kereséssel kapott pozíciója, az eredeti képhez hasonlóan az utolsó konvolúciós réteg jellemzőtérképeihez vannak rendelve. Ezután a javított jellemzőket a többszintű SPP összevonja, és minden ablakhoz létrehoz egy adott hosszúságú jellemzővektort. Az SPP-Net képes megőrizni mind a globális, mind a helyi információit a képnek, szóval nagyobb pontossággal rendelkezik, mint az R-CNN. [21]

A Fast R-CNN az SPP réteg alapján létrehozta a Region of Interest (RoI) összevonási réteget, amely egy egyszintű SPP rétegnek tekinthető és képes különböző méretű képeket is feldolgozni. A Fast R-CNN módszer az SPP-Nethez képest a következő előnyökkel rendelkezik:

* A Fast R-CNN egy végponttól végpontig tartó objektum felismerési rendszert tanít, az SPP pedig egy többlépéses folyamatot.
* A Fast R-CNN egyszerre tanítja az osztályozást és a határoló dobozok létrehozását.
* Az SPP-net nem tudja frissíteni a konvolúciós rétegeket tanítás közben, a Fast R-CNN azonban képes frissíteni az összes hálózati réteget. Az objektumdetektálási feladathoz a konvolúciós rétegek paramétereinek finomhangolása is szükséges és fontos, mert a megelőző konvolúciós rétegek több helyinformációt tudnak megőrizni, mint a későbbiek. [21]

A régiójavaslatok kiszámítása még így is egy bökkenőt jelent a Fast R-CNN szempontjából. A minőségi régiójavaslatok nagyban befolyásolják az objektum felismerés gyorsaságát és pontosságát. Ennek megoldására találták ki a régiójavaslati hálózatot (Region Proposal Network), amely szinte valós időben képes észlelni az objektumokat. A régiójavaslati hálózat egy teljesen konvolúciós hálózat (Fully Convolutional Network) amely egyidőben jósolja meg az objektumok határait és az objektivitási pontjukat minden lehetséges helyen. Az alapötlete az, hogy a CNN által magas visszahívású régiójavaslatokat generál, amely nemcsak felgyorsítja a felismerési sebességet, hanem a kombinálni lehet a CNN végpont alapú tanításával. [21]

A Faster R-CNN egyesítette a régiójavaslatot és a CNN osztályozását egy végpontok közötti detektálási hálózathba, javítva ezzel mind a sebességet, mind a pontosságot. A Faster R-CNN azonban még mindig nem képes nagy pontossággal teljesíteni a valós idejű objektum-észlelés követelményeit, mivel a CNN jellemzői kivonásának kiszámítása még mindig időigényes. [21]

Összességében az R-CNN, az SPP-Net, a Fast R-CNN és a Fater R-CNN mind a CNN-en alapuló objektum felismerést fejlesztették tovább, hogy gyorsabb és pontosabb módszert kapjanak. Az R-CNN módszerek jelenleg is az objektum felismerés egyik legfőbb ágát képviselik. [21]

## Körök keresése

A felismert hangjegyek „fejeinek” detektálására a Hough Transform ideális megoldást nyújt, azonban ennek megelőző lépéseként a képen egy éldetektáló algoritmust kell lefuttatni.

### Canny

A Canny egy éldetektáló operátor, amely többlépcsős algoritmust használ a képek éleinek széles skálájának észlelésére. John F. Canny fejlesztette ki 1986-ban. A Canny egy olyan technika, amely hasznos szerkezeti információkat nyer ki a különböző objektumokból, és jelentősen csökkenti a feldolgozandó adatok mennyiségét. Széles körben alkalmazzák különféle gépi látás rendszerekben.[2][3][20]

Az éldetektálási folyam öt különböző részre bontható:

* Gauss szűrő használata:

Ennek a lépésnek a zajcsökkentés, illetve a kép simítása a célja. Az algoritmus egy Gauss-szűrő kernelt von össze a képpel, ezzel csökkentve az éldetektálóra ható zaj hatását. Egy kétdimenziós Gauss szűrő esetében az algoritmus a következő:

(12)

* A kép intenzitási gradienseinek megtalálása:

A kép élei többféle irányba mutathatnak, ezért a Canny algoritmus négy szűrőt használ az elmosódott kép vízszintes, függőleges és átlós éleinek észlelésére. Az elsőrendű parciális deriváltak meghatározhatók a következő egyenletekkel:

(13)

(14)

Az éldetektáló függvény az első derivált értékét adja vissza vízszintes és függőleges irányban, ez alapján meghatározható az él gradiense és iránya:

(15)

(16)

* Nem maximális szűrés(non-maximum suppression):

Ez az algoritmusok egy osztálya, amely egy entitást (pl. éleket) választ ki a sok átfedő entitás közül. Megválaszthatjuk a kiválasztási kritériumokat, hogy elérjük a kívánt eredményt. A kritériumok leggyakrabban a valószínűségi szám valamilyen formája és az átfedés mértéke. Ez a lépés megszűri az esetlegesen detektálandó éleket, így a jövőben ez kevesebb erőforrás igénnyel jár.[2][3][20]

* Kettős küszöbértékkel végzett élmeghatározás:

A nem maximális szűrés alkalmazása után a fennmaradó élpixelek pontosabb képet adnak a képen található valódi élekről. Néhány élpixel azonban megmarad, amit a zaj és a színváltozás okoz. Ezeknek a hamis élpixeleknek a figyelembevételéhez elengedhetetlen a gyenge gradiens értékű élpixelek kiszűrése és a magas gradiens értékű élpixelek megőrzése. Ehhez meghatározunk egy maximum és egy minimum küszöbértéket. Ha egy élpixel gradiens értéke magasabb, mint a maximum küszöbérték, akkor erős élpixelként jelöli meg, ha egy élpixel gradiensértéke kisebb, mint a maximum küszöbérték, de nagyobb, mint a minimum küszöbérték, akkor gyenge élpixelként jelöli meg, abban az esetben pedig, ha egy élpixel gradiens értéke kisebb, mint a minimum küszöbérték, akkor azt a rendszer kiszűri. A két küszöbérték empirikusan meghatározott, és az adott bemeneti kép tartalmától függ.[2][3][20]

* Élek és élkapcsolatok detektálása:

Az erős pixelnek érzékelt élpixeleket minden bizonnyal megtartjuk, azonban a gyenge pixeleknél érdemes megvizsgálni, hogy kapcsolódnak-e egy erős élpixelhez. Az élkapcsolat nyomon követéséhez blob-elemzést alkalmazunk egy gyenge élpixel és a hozzá kapcsolódó 8 szomszédos képpont alapján. Mindaddig, amíg van egy erős élpixel, amely szomszédos a gyenge élpixellel, addig ez a gyenge élpont azonosítható olyanként, amelyet meg kell őrizni.[2][3][20]



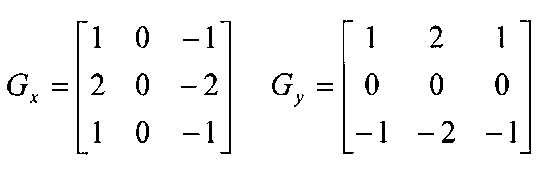
12. ábra: Példa Canny-vel végzett éldetektálásra

Forrás: https://miro.medium.com/max/1254/1\*ZCyKWsmDoj6V-dNwKlKxyA.png (megtekintve: 2022.04.26.)

Habár a Canny a legszélesebb körben ismert és alkalmazott eljárás az éldetektálásra, érdemes megemlíteni, hogy több különböző alternatívája is létezik.

* Sobel módszer:

A sobel az egyik leggyakrabban használt éldetektáló módszer a Canny mellett. A kép egy kicsi, szétválasztható és egész értékű szűrővel történő vízszintes és függőleges irányú konvolúción alapul, ezért viszonylag kevés számítási feladattal jár. A Sobel éljavító szűrő előnye, hogy egyidejűleg biztosítja a megkülönböztetést, ami az élreakciót adja és a simítást, ami csökkenti a zajt.[2][3][20]

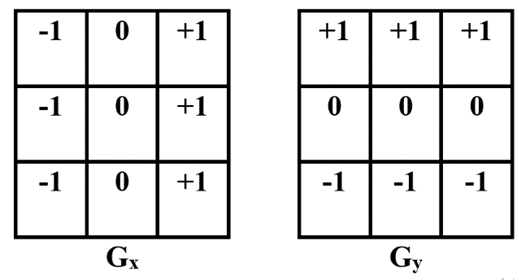


13. ábra: A Sobel Mask kerneljei az éldetektáláshoz

Forrás: <https://nikatsanka.github.io/images/sobel-mask.png> (megtekintve: 2022.05.03.)

* Prewitt módszer:

A Prewitt operátor hasonló a Sobel operátorhoz, és a képek függőleges és vízszintes éleinek észlelésére szolgál. A Sobel-lel ellentétben azonban ez az operátor nem helyez hangsúlyt a maszk közepéhez közelebb eső pixelekre.[2][3][20]

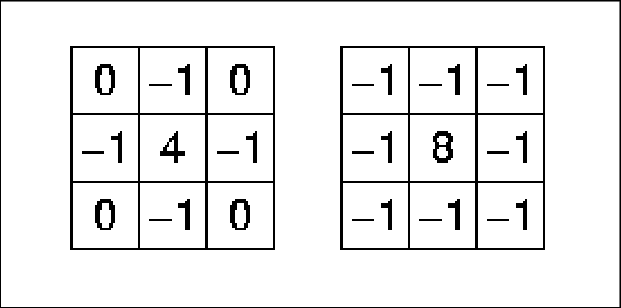


14. ábra:Prewitt Mask kerneljei az éldetektáláshoz

Forrás: <https://www.researchgate.net/profile/S-N-Kumar/publication/317754223/figure/fig3/AS:565335482351616@1511797890242/Masks-for-the-Prewitt-gradient-edge-detector-The-Laplacian-operator-is-based-on-second.png> (megtekintve: 2022.05.03.)

* Laplacian módszer:

A Laplacian némileg eltér az eddig tárgyalt módszerektől. A Sobel és Prewitt éldetektoraival ellentétben a Laplacian éldetektor csak egy kernelt használ. Egy menetben számítja ki a másodrendű származékokat. Két gyakran használt kis kernel:



15. ábra: Laplacian operátor 2 féle kernele éldetektáláshoz

Forrás: <https://www.researchgate.net/profile/Siti-Yasiran/publication/261459927/figure/fig1/AS:650032764170264@1531991296526/Two-commonly-used-discrete-approximations-to-the-Laplacian-filter.png> (megtekintve: 2022.05.03.)

Mivel ezek a maszkok egy második derivált mérést közelítenek a képen, nagyon érzékenyek a zajra. Ennek kijavítására a képet gyakran Gauss-féle simítással simítják ki a Laplacian-szűrő alkalmazása előtt.[2][3][20]

### Hough Circle Transform

A Hough Cirlcle Transform a Standard Hough Transform azon változata, amely kör alakzatokat képes meghatározni egy adott képen. Működése némileg hasonló a korábban már említett Hough Line Transforméhoz, azonban bizonyos dolgokban eltér tőle. A kört valójában egyszerűbb a paramétertérben ábrázolni, mint a vonalat, mivel a kör paraméterei közvetlenül átvihetők a paramétertérbe. A kör egyenlete:

(15)

Mint látható, a körnek három paramétere van, a, b és r. Ahol a és b a kör középpontja x és y koordináta alapján, az r pedig a sugár. A kör parametrikus ábrázolása:

(16)

(17)

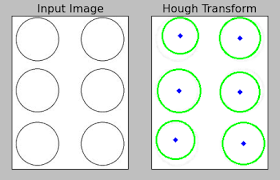
[2][18]

A Hough Circle Transform használatával történő körkeresés folyamata a következő:

Először megkeressük a képen az összes élt. Ennek a lépésnek semmi köze a Hough Transform-hoz, és bármilyen éldetektáló technika használható. Ebben a projektben ezt a Canny segítségével hajtottam végre.

Ezután lefut a kereső algoritmus, amely minden élpontban rajzol egy kört, amelynek középpontja a kívánt sugarú pontban van. Ezt a kört a paramétertérbe rajzolja oly módon, hogy az x tengely az ’a’ érték, az y tengely a ’b’ érték, míg a z tengely a sugár. A megrajzolt kör kerületéhez tartozó koordinátákon megnöveli az akkumulátor mátrixunkban található értéket, amely lényegében megegyezik a paramétertérrel. Ily módon a bemeneti kép minden élpontján átsöpri a kívánt sugarú köröket, és növeli az értékeiket az akkumulátorban.

Az akkumulátor ekkor az egyes koordinátákon áthaladó körök számának megfelelő számokat fog tartalmazni. Így a legmagasabb számok a képen látható körök középpontjának felelnek meg.[2][18]



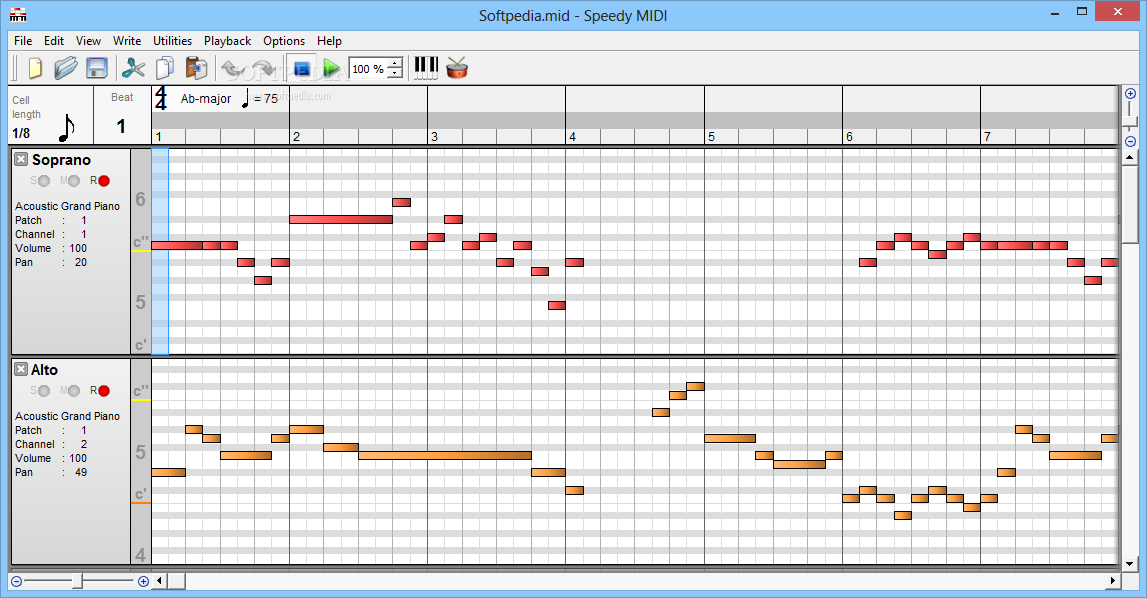
16. ábra: A Hough Circle Transform eredménye egy tesztképen

Forrás: <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQ-yYe4UHpJptYlLS7WkIEI3xZmfP-nV3nTMw&usqp=CAU> (megtekintve: 2022.04.26.)

## Kimenet generálása

A bemenetként kapott kottát a program lefutásának végén egy MIDI outputtá alakítja a szoftver. Ennek végrehajtásához egy pretty\_midi nevű függvénykönytárat használok. A pretty\_midi segédfunkciókat/osztályokat tartalmaz a MIDI adatok kezelésére, így azok olyan formátumban vannak, amelyből könnyen módosítható és kinyerhető az információ.

A pretty\_midi modult MIDI-fájlok létrehozásához, kezeléséhez és elemzéséhez hozták létre. Célja, hogy a MIDI adatokra alkalmazott leggyakoribb műveleteket a lehető legegyszerűbbé tegye. A pretty\_midi egy MIDI fájlt képvisel az alábbi ábrán látható hierarchikus módon. A modul MIDI fájlok elemzésére és írására, MIDI adatok létrehozására és manipulálására, szintézisre és információ kinyerésére szolgáló funkciókat tartalmaz. Forráskódja szintén könnyen érthető és módosítható, és elérhető a GitHubon. A következő két részben a pretty\_midi tervezését és használatát ismertetem. [19]



17. ábra: MIDI file felépítése és kinézete

Forrás: <https://windows-cdn.softpedia.com/screenshots/Speedy-MIDI_1.png> (megtekintve: 2022.04.26.)

Amint az ábrán látható, a pretty\_midi a MIDI-adatokat osztályok hierarchiájaként jeleníti meg. A tetején a PrettyMIDI osztály található, amely olyan globális információkat tartalmaz, mint a tempóváltások és a MIDI felbontás. Tartalmazza a hangszerek listáját is. Minden hangszert egy programszám és egy zászló határoz meg, amely jelzi, hogy melyik adott hangszerről van szó. A hangszerosztály példányai három listát is tartalmaznak, egy-egy listát a hangjegy (Note), a hajlítás (PitchBend) és a vezérlés (ControlChange) osztálypéldányokhoz. A Note osztály a MIDI hangok tárolója, sebesség, hangmagasság, valamint kezdési és befejezési idő attribútumokkal. Hasonlóképpen, a PitchBend és a ControlChange osztályok egyszerűen rendelkeznek attribútumokkal a hajlítás vagy a vezérlés változásának idejére és értékére.[19]

A legfelső szintű PrettyMIDI osztály példányosítható egy létező MIDI fájl elérési útjával, ebben az esetben az osztály a fájl elemzése által lesz feltöltve. Előzetes fájl nélkül is példányosítható, a MIDI adatok létrehozásához a semmiből. A kimenethez a PrettyMIDI osztály rendelkezik egy írási funkcióval, amely az adatait egy érvényes MIDI fájlba exportálja.[19]

# A program megvalósítása

Az elkészítendő program részletes megtervezése után nincs más hátra, mint implementálni a korábban leírtakat. A megvalósításhoz a programozási nyelven kívül (amelyet már korábban részleteztem) szükség van még egy fejlesztői környezetre is. Az én választásom a Visual Studio Code-ra esett, amely egy ingyenes, nyílt forráskódú fejlesztői környezet, mely számos operációs rendszeren elérhető és beépített Git támogatással is rendelkezik. Ezen kívül pedig rengeteg programozási nyelvet támogat, tehát nagyszerűen használható például Python, C++, Java, C#, PHP, SQL és egyéb programok írására.

## Előkészületek

A projekt első lépéseként két függvénykönyvtárat kell letöltenem, majd beimportálnom ahhoz, hogy használni tudjam a funkcióikat. Ezek nem mások, mint az OpenCV és a NumPy. Ezután következik a feldolgozandó kép beolvasása. Ehhez a művelethez először szükségem van néhány képre, amelyeken a tesztelést el fogom végezni. Az interneten számos zenei kotta érhető el jó minőségben, de talán a legnagyobb gyűjtemény a ChoralWiki oldalon található meg, ahol több ezer zenei kotta található meg rendszerezett formában, és tölthető le PDF, Midi, MusicXML és egyéb formátumokban. Mivel számomra valamilyen képformátumú bemeneti adatra van szükségem, így PDF formátumban töltöttem le őket, majd egy internetes konverter segítségével JPG formátumra alakítottam őket. Így már felhasználhatók bemenetként a digitalizáló programhoz.

## Küszöbérték

A könyvtárak importálása és a kép beolvasása után a küszöbérték (threshold) megadása következik. A beolvasott képet először szürkeárnyalatos képpé kell átalakítani. Ez egyszerűen elvégezhető, hiszen az OpenCV-ben van erre egy beépített metódus, ami az RGB képeket szürkeárnyalatossá teszi. Miután ez elkészült egy bináris képet kell létrehozni ebből a képből, amelyet küszöbérték megadással kapok meg.

## Kottavonalak megkeresése

A zenei kottákban fellelhető hosszú vonalakat, amelyeken a hangjegyek is elhelyezkednek, kottavonalaknak hívjuk. A kottaírásban jellemzően őt kottavonal alkot egy sort. A vonalak bizonyos hangmagasságokat jelölnek, és a hangokat jelölő hangjegyek pedig ezeken a vonalakon, vagy a vonalak között helyezkedhetnek el.

Ahhoz, hogy a program megtalálja és felismerje ezeket a kottavonalakat, a Probabilistic Hough Transformot alkalmazom. Az ehhez szükséges eljárás a HoughLinesP(), amelynek miután megadtuk a szükséges paramétereit, azaz a képet, amelyen a keresést végzi, a rho-t, a theta-t, a minLineLenght-et és a maxLineGap-et. A korábbiakban ismertetett módon ez az algoritmus elkezdi a képen található adott hosszúságú és töredezésmentes vonalakat megkeresni és felismerni. Ahhoz, hogy jól működjön az eljárás, a bináris képen kell lefuttatni. Miután elkészült a vonalak megkeresésével, a vizuálisabb ellenőrzés érdekében kirajzoltatom a kottavonalakat, amelyeket a HoughLines megtalált.



18. ábra: Vonalfelismerés eredménye

Ezeket a vonalakat eltárolom egy adott listába, hogy a későbbiekben felhasználhatóak legyenek. Ezt a listát egyszerűen csak Lines-nak neveztem el, hogy minél egyértelműbb legyen. Azonban az ebben eltárolt vonalak nem mindig feleltek meg az elvárásoknak, esetenként nem talált meg vonalakat, vagy több vonalat talált meg, mint kellene.

Ezen problémák kiküszöbölésére hoztam létre még három listát, amelyek segítségével javítok a teljes vonalkeresési procedúra sikerességén. Először is x irányú, azaz vízszintes hosszúság szerint sorba rendezem a detektált vonalakat, ás azokat ilyen formában lementem egy goodLines listába. Ezt követően megvizsgálom, hogy van-e szinte azonos pixelhelyen több vonal is, ha akad ilyen, azokat összevonom, hiszen azok nagy valószínűséggel egy és ugyanaz vonal, csak a vastagsága miatt az algoritmus több vonalnak értelmezte (például egy 5 pixel vastag vonalat érzékelhet 5 darab 1 pixeles vonalnak). Ezt a gyakorlatban úgy valósítottam meg, hogy a goodLines listából egyesével hozzáadtam a vonalakat egy tempLines listába, amelyben megvizsgáltam az adott kottavonal y koordinátáját, majd minden esetben összevetem a következő vonal koordinátájával és ha a kettő közötti különbség kevesebb, mint adott pixel, akkor összevontam a két vonalat.

Amikor ezzel elkészültem, a vonalakat a tempLines listából hozzáadtam a permaLines listához, majd mivel y koordináta szerint vannak rendezve, ötösével eltárolom őket, így a kottasorok egyben vannak eltárolva.

## Kottavonalak kitörlése

Ahhoz, hogy a hangjegyeket könnyebben fel tudjam ismerni, a korábban megtalált és megjelölt vonalakat eltávolítom a képről. Így megelőzöm azokat a problémákat, amelyekkel akkor szembesülnék, ha úgy akarnám felismerni a hangjegyeket, hogy a háttérben a vonalak bezavarnának a felismerési algoritmusnak. A vonalak eltávolítását morfológiai szűrő segítségével hajtom végre.

Annak érdekében, hogy a vonalakat ne az eredeti képről távolítsam el – ezzel utólag használhatatlanná téve a képet – először készítek egy másolatot a képről, amelyen lefuttatom a programot. Ezután egy 1 pixel magas 5 pixel széles, téglalap alakú, csupa 1-es értékkel ellátott strukturáló elemmel eróziót végzek a képen, majd ezt követően egy tágulást, azaz egyszerűbben kifejezve egy nyitást hajtok végre. Mindez után a bitwise\_not() függvény segítségével megfordítom a pixelek küszöbértékeit, azaz ami 0 volt az 255 lesz és fordítva. Így a végeredménynél egy szemléletesebb képet kapok.



19. ábra: A kotta kinézete a kottavonalak eltávolítása után.

## Alakzatok kontúrjainak megkeresése és megjelölése

A kottavonalak kitörlése után egy-két segédvonaltól és egyéb alakzattól eltekintve már csak a számunkra fontos, felismerni kívánt elemek maradtak a kottában. Így itt az ideje megkeresni ezeknek az objektumoknak a kontúrjait, hogy mind alakjukat, mind elhelyezkedésüket meg tudjam határozni későbbi felhasználás céljából.

Első lépésként invertálom az előbbi képet, vagyis fehér alapon fekete helyett fekete alapon fehérré alakítom. Erre azért van szükség, mert a findContours() függvény fehér objektumokat keres fekete háttéren. Mivel előre tudom, hogy nem kell arra az eshetőségre gondolnom, hogy egyik alakzat egy másik alakzatban helyezkedik el, így a RETR\_EXTERNAL visszakeresési módot alkalmazom, amely csak a legkülső kontúrokat adja vissza. Közelítési módnak pedig a CHAIN\_APPROX\_NONE metódust adom meg, amely a megtalált alakzat összes határpontját eltárolja. Ez a módszer növelhet a hatékonyságon, azonban több memóriát is fogyaszt.

Ezután a bináris képet színessé alakítom, hogy az eredmény látványosabb legyen, majd a drawContours() függvénnyel kiemelem az objektumot kontúrjait a számomra tetsző színben és vonalvastagságban.



20. ábra: Kontúrkeresés eredménye

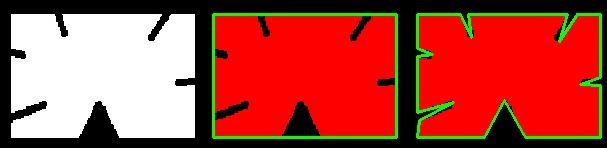
## Alakzatok eltárolása

Most, hogy az objektumok körvonalait megjelöltem, következő lépésként eltárolom az alakjukat és koordinátáikat. Ehhez létrehozok egy tömböt, amelyben ezeket a szimbólumokat el fogom tárolni.

Egy for ciklus segítségével végig iterálok a korábban megrajzolt kontúrokon, majd az arcLength() függvény segítségével meghatározom a kontúrok kerületét. A függvény két argumentumot vár tőlünk, az első maga a kontúr, amit eltároltunk, a második pedig azt határozza meg, hogy maga az alakzat zárt körvonalú-e vagy csak egy görbe. Az én esetemben zárt alakzatokról beszélünk, így ennek az argumentumnak true értéket adtam.

Miután ezzel megvagyok, az approxPolyDP() függvényt alkalmazom. Az algoritmus célja egy vonalszakaszokból álló kontúr (amelyet vonalláncnak is neveznek) alapján egy hasonló, kevesebb ponttal rendelkező kontúrt találjon. Az algoritmus az eredeti kontúr és az egyszerűsített kontúr közötti maximális távolság (azaz az epszilon) alapján határozza meg a különbséget. Az egyszerűsített kontúr az eredeti kontúrt meghatározó pontok részhalmazából áll.

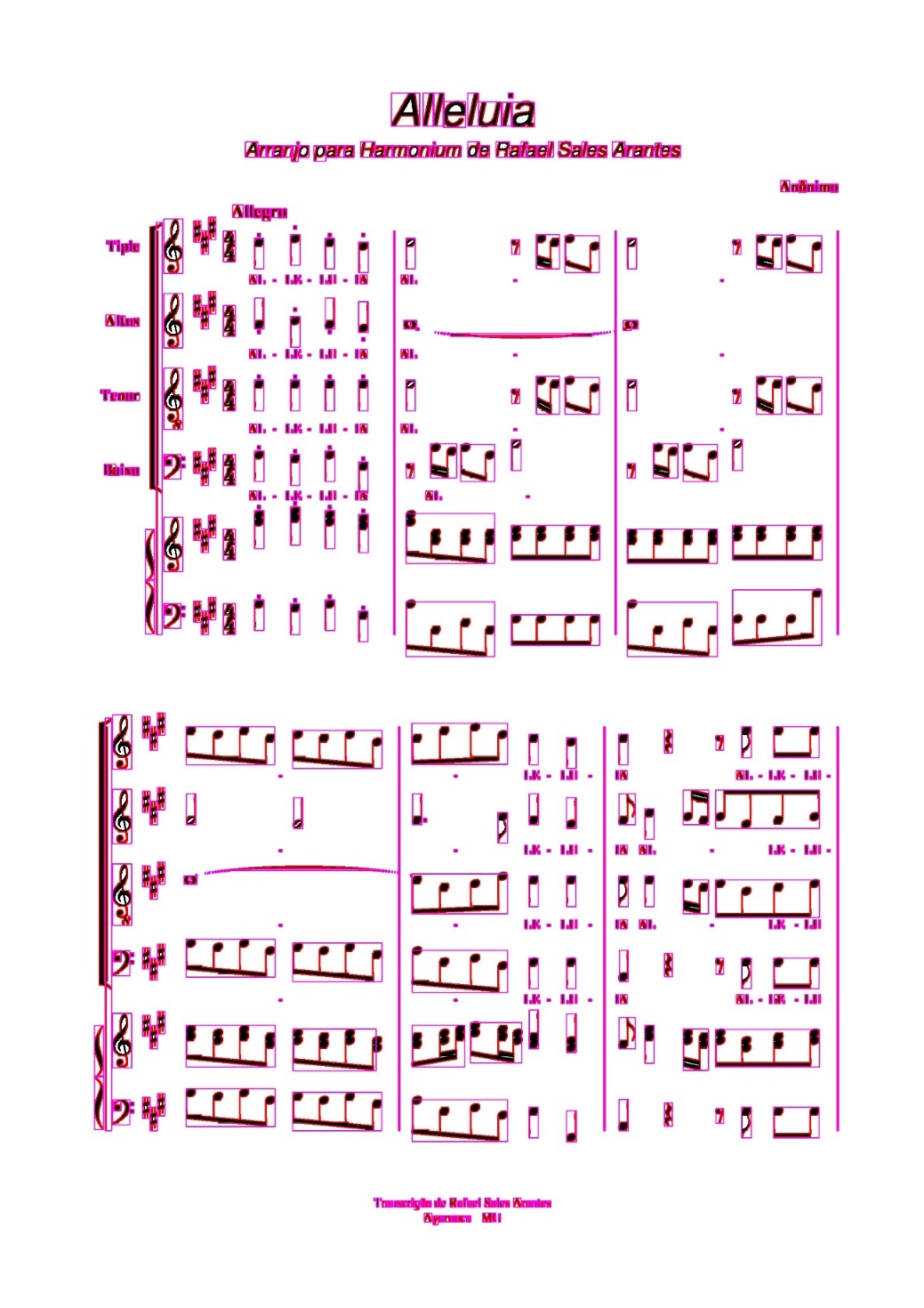
Az algoritmus 3 értéket fogad, az első az eredeti kontúr, a második az epszilon, vagyis a közelítési érték, a harmadik pedig, hogy zárt alakzatról van-e szó. Az alábbi képen középen egy 10%-os közelítési érték eredménye látható, míg jobb oldalon egy 1%-os.



21. ábra: Különböző epszilonokkal végzett közelítés

Forrás: <https://opencv24-python-tutorials.readthedocs.io/en/latest/_images/approx.jpg> (megtekintve: 2022.03.18.)

Az approxPolyDP() lefutása után egy határoló téglalapot (bounding rectangle) hozok létre, hogy a számomra érdekes területeket (area of interest) kiemelhessem. Mindezt a boundingRect() függvénnyel tudom megtenni, amely egy hozzávetőleges téglalap létrehozására szolgáló függvény, ami az alakzatok kontúrjai alapján kiemeli a kívánt részeket. A boundingRect() négy numerikus értéket ad vissza, amikor a kontúrt argumentumként adjuk át. Ez a 4 érték az x, y, w, h értékeknek felel meg. Ezek az x koordinátát, az y koordinátát, a szélességet és a magasságot jelölik. Ezek az értékek használhatók a téglalap megrajzolására vagy egy képrész kivágására a pixelkoordináták segítségével. Az algoritmusnak öt argumentumot kell megadni bemenetként, a képet, amin a kirajzolás végbemegy, a kiindulási pont, a végpont, a téglalap színe és a vonal vastagsága.



22. ábra: Határoló téglalapok megjelenítve a kottán

Ezután eltárolom a határoló téglalapok által körbejelölt szimbólumokat egy symbols nevű tömbbe, majd a sorted() algoritmussal szélesség alapján sorba rendezem az elemeket, így egymás után kerülnek az azonos szimbólumok.

## Neurális modell létrehozása és tanítása

Most, hogy a kotta elemei pozíciójukkal és alakjukkal együtt el vannak mentve külön elemekként, itt az ideje, hogy beosztályozzuk ezeket az objektumokat, azaz felismertessük őket a neurális hálózattal. Mint korábban említettem egy CNN-t, vagyis egy konvolúciós neurális hálózatot hozok létre ennek a feladatnak az elvégzésére. Ahhoz azonban, hogy létre tudjak hozni egy CNN-t be kell importálnom kettő nagyon fontos függvénykönyvtárat. Ezek a TensorFlow és az abban megtalálható egyik API, a Keras.

A TensorFlow egy gépi tanulási rendszer, amelyet széles körben alkalmaznak és heterogén környezetekben működik. Számítási modellje változó állapotú adatfolyam-gráfokon alapul. A gráfcsomópontok leképezhetők különböző gépekre, és az egyes gépeken belül CPU-kra, GPU-kra és egyéb eszközökre. A TensorFlow számos alkalmazást támogat, de különösen a mély neurális hálózatokkal való tanítást és következtetéseket célozza meg. Platformként szolgál a kutatáshoz és a gépi tanulási rendszerek bevezetéséhez számos területen, mint például a beszédfelismerés, a gépi látás, a robotika, az információkeresés és a természetes nyelvi feldolgozás. [1]

A Keras egy kompakt és könnyen megtanulható, magas szintű Python-könyvtár a mély tanuláshoz, amely a TensorFlow mellett fut. Lehetővé teszi a fejlesztők számára, hogy a mély tanulás főbb funkcióira összpontosítsanak, például rétegek létrehozására neurális hálózatokhoz, miközben ügyel a tenzorok apró részleteire, alakjukra és matematikai részleteire. A TensorFlow-nak a Keras hátterének kell lennie, de használhatjuk a Keras-t mély tanulási alkalmazásokhoz anélkül, hogy interakcióba lépnénk a viszonylag összetett TensorFlow-val. A keretrendszernek két fő típusa van: a szekvenciális API és a funkcionális API. A szekvenciális API a rétegek sorozatos egymás utániságán alapul, ez a Keras leggyakoribb használata és a Keras legegyszerűbb része. A szekvenciális modell lineáris réteghalmaznak tekinthető. [12][13]

Ahhoz, hogy a szekvenciális modellünk megfelelően működjön, először be kell tanítani. Ehhez mintaképekre van szükség egy kottán megtalálható elemekről, amelyek típusonként külön vannak szedve, így meg tudja különböztetni a különböző komponenseket a modellünk, amikor felismertetjük vele a tanításra szánt képeket. A tanítási képeken kívül validáló képekre is szüksége lesz a modellnek, amelyekkel a becslések során önmagát tudja ellenőrizni, és a becslés sikerességét követni. Ezek a képek ugyanúgy csoportonként külön mappában vannak tárolva, azonban az fontos, hogy ne pontosan ugyanazok a képek legyenek a validáló képek, mint amelyeken a tanítást végezzük, mert ebben az esetben a becslés csalóka eredményt adhat.

Ezen képek összességét hívjuk datasetnek, vagyis adatkészletnek. Az interneten számos dataset megtalálható, amely zenei kották elemeit tartalmazza, én is ezek közül választottam egyet, név szerint a Rebelo Dataset-et.

Miután megvan az adatkészlet, megadom a programnak, az elérési utat a képekhez, amelyekre a modellnek szüksége van. Ezután bizonyos előzetes feldolgozásokat hajtok végre a tanítási és a validációs adatkészleteken. Először megadom neki, hogy melyik mappából használja a képeket, majd a köteg méretét, amely a különböző fajta felismerendő elemeknek a száma. Mindezek után átméretezem a képeket 32pixel szélességűre és magasságúra, mert a modellt úgy állítom be, hogy ekkora méretű képekkel dolgozzon. A képeket ezután szürkeárnyalatossá alakítom, majd összekeverem az adatkészletben a képeket, hogy a modell ne sorrendben kapja meg azokat, így nehezebb lesz a felismerése. Végül beállítom, hogy a tanítás során a készlet mekkora részét használja validálásra.

Mielőtt a modellnek odaadom a képeket tanításra, létrehozok egy másik szekvenciális modellt, amely a data augmentationt, vagyis a képek véletlenszerű átalakítása. Erre annak érdekében van szükség, hogy a modell a kisebb eltérésekkel rendelkező, de azonos osztályba tartozó egyedeket is képes legyen felismerni, mivel erre is betanítjuk ily módon. Három transzformációt hajtok végre a képeken véletlenszerűen a tanítás előtt, ezek a közelítés, elforgatás és az eltolás. Mindegyiknek adok egy 0-1 közötti értéket, amely százalékos értéket jelent.

model = Sequential()

model.add(data\_augmentation)

model.add(layers.Input(shape=(120,60,1)))

model.add(Conv2D(8, 3, activation="relu"))

model.add(MaxPool2D(2))

model.add(Conv2D(16, 3, activation="relu"))

model.add(MaxPool2D(2))

model.add(Conv2D(32, 3, activation="relu"))

model.add(MaxPool2D(2))

model.add(Conv2D(64, 3, activation="relu"))

model.add(MaxPool2D(2))

model.add(Dropout(0.5))

model.add(Flatten())

model.add(Dense(64, activation='relu'))

model.add(Dense(32, activation='relu'))

model.add(Dense(10, activation='softmax'))

1.kódrészlet: A feladathoz felépített szekvenciális CNN modell

A modell definiálása és a rétegek egymásra rakása után konfigurálni kell a modellt. Ez a konfigurációs folyamat a fordítási fázisban megy végbe. A modellt betanítás előtt le kell fordítani, és meg kell határozni a veszteségfüggvényt, az optimalizálókat és az becsléshez szükséges mérőszámokat. Mindezt a model.compile() függvény segítségével teszem meg.

A következő lépés pedig maga a modell betanítása. Ennek megvalósítására a model.fit() metódust hívom meg. Ez fogja végrehajtani a modell betanítását, a neki megadott paraméterek alapján. Vannak kötelezően megadandó paraméterek, de vannak opcionálisak is. Meg kell neki adni mind a tanításra szánt, mind a validációra szánt adatkészletet, majd megmondani neki, hogy hány alkalommal fusson le, végül beállítani, hogy milyen formában jelenítse meg nekünk a tanítás eredményét. Ezt az én esetemben úgy állítottam be, hogy mindegyik tanítás egy külön sorban jelenjen meg. Ennek a modellnek ezeket a paramétereket állítottam be.

A tanítás végeztével a model.save() függvénnyel elmentem a szekvenciális modellt egy általam választott néven.

## Neurális modell tesztelése

A modell tesztelése a következő lépés a betanítás után. Ilyenkor dől el, hogy a felépített modell mennyire működik hatékonyan. Habár ez nem csak a modellen múlik, az adatkészlet is nagyban hozzájárul, valamint a felismerésre szánt objektumok is.

Az első lépésnél megadom a felismerni kívánt objektumról készült képet, majd bináris képpé alakítom. Amint ezzel készen vagyok, átméretezem a tesztképet akkorára, amekkora méretre a neurális modell be lett tanítva, esetemben ez 32x32-es méretet jelent. Ezután a models.load\_model() metódussal betöltöm a korábban létrehozott modellemet. A model.predict() függvény fogja elvégezni az objektumok típusának becslését, majd az eredményt a konzolra kiíratom, így tudom ellenőrizni, hogy megfelelő értéket ad-e vissza a modell.

## Neurális modell implementálása a programba

Hasonlóan a neurális modell teszteléséhez, be kell tölteni először a programba a modellt az előbb ismertetett módon. Itt létrehozok egy classified\_symbols nevű tömböt, ahová a már beosztályozott elemeket fogom eltárolni. Egy for ciklussal végigiterálok a symbols tömbön, amelybe korábban eltároltam a kottából kivágott objektumokat és mindegyiket átméretezem 32x32-es méretre. Végrehajtom rajtuk a predikciót, majd a koordinátáikkal és a típusukkal együtt – amelyet a modell meghatározott – eltárolom őket az újonnan létrehozott tömbbe. Ezzel előkészítettem a szoftvert arra, hogy képes legyen egy kívánt kimenetet generálni az eddig eltárolt adatokból.

## Kimenet generálása

A szoftver megvalósításának fő célja, hogy a zenei kottát, amelyet bementként megkapott, a folyamatok lefutásának végén egy olyan kimenetté alakítsa, amit egy kottaolvasó szoftver képes feldolgozni. Ezek a szoftverek általában valamilyen zenei fájlt várnak bemenetként, így az én választásom is egy ilyen fájlra esett, egészen pontosan egy MIDI fájl generálására. A kimenet generálása több lépésből áll, a tényleges konvertálás előtt még néhány előzetes lépés végrehajtása szükséges.

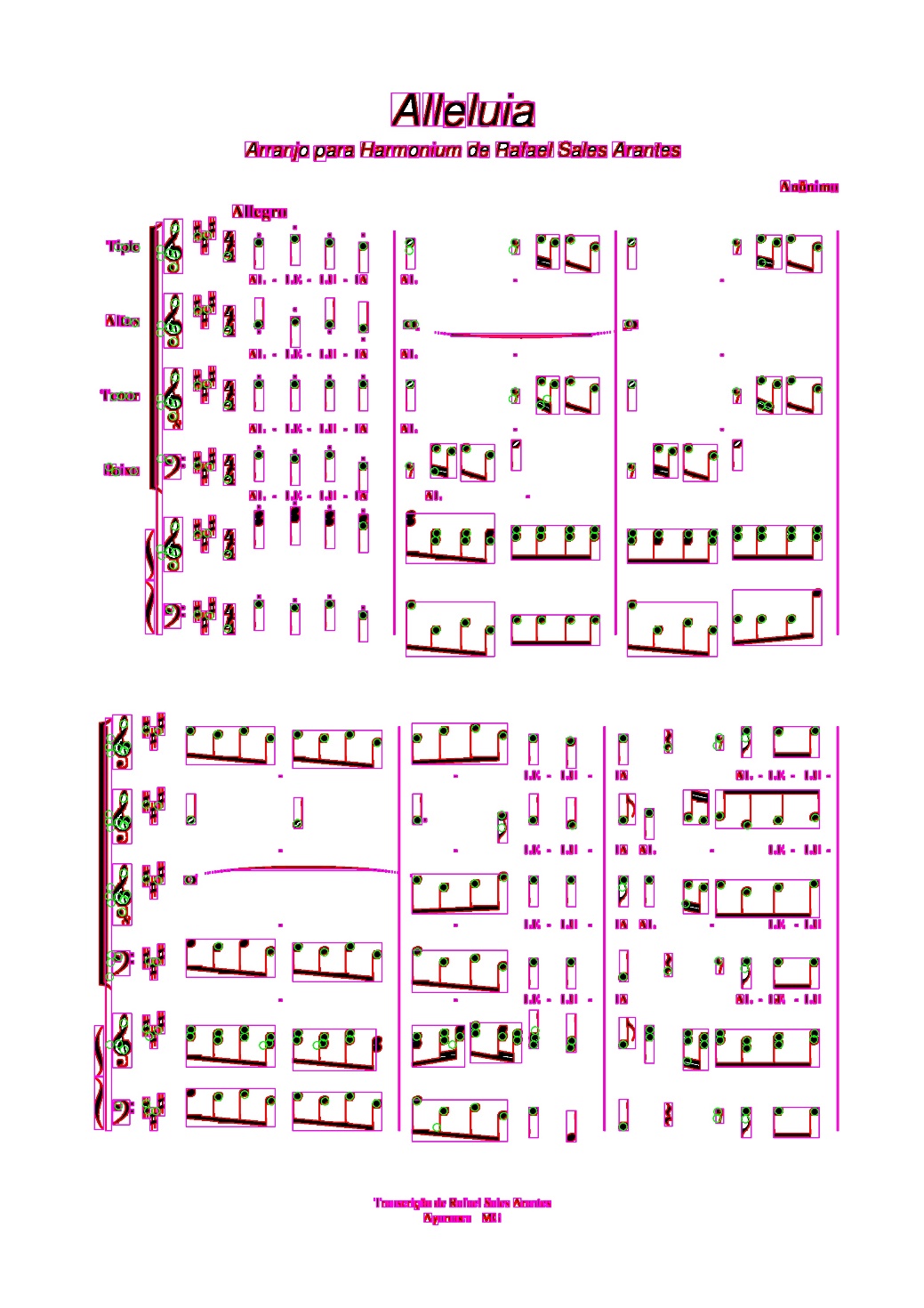
### Hangjegyek fejeinek megkeresése

Első lépésként egy Canny éldetektálást alkalmazok a bemeneti kép szürkeárnyalatos verzióján. Ezt a cv2.Canny() függvénnyel tudom megtenni. Három paramétert adok át a metódusnak, a képet, amelyen az éldetektálást elvégzi, illetve a minimum és a maximum küszöbértéket. A további lépéseket ezen az átalakított képen fogja a program elvégezni.

Ezután meg kell határoznom a kottavonalak közötti távolságot, mert erre az információra is szükség van ahhoz, hogy a hangjegyek fejeinek elhelyezkedését meg tudjam állapítani. Mivel a vonalak 5-ös csoportokban le vannak tárolva egy korábbi eljárásnak köszönhetően, így a vonalak közti távolságot könnyen ki lehet számolni úgy, hogy az egyik vonal y koordinátáját kivonom a felette lévő vonal y koordinátájából és a kapott értéket eltárolom egy változóban.

Miután ez készen van, a cv2.HoughCircles() algoritmus segítségével köröket detektálok a képen, mivel a hangjegyek részét képező pontok ugyan nem szabályos körök, de kellően hasonló alakzatok ahhoz, hogy felismerje őket ez a függvény. A metódus paraméterként megkapja az imént generált képet, a keresési metódust, ami esetemben gradiens alapú, a kép skálázásának nagyságát, a körök középpontjai közti minimális távolságot, amit az előbb kiszámolt vonaltávolság alapján adok meg, két küszöbértéket, amelyek a Cannyhez hasonlóan a minimum és a maximum értékek lesznek, valamint a körök minimális és maximális sugarát. A sugár megadásához is a korábban megadott vonaltávolság használható, hiszen tudjuk, hogy jó esetben a hangjegyek fejei nagyjából akkora átmérőjűek, mint a két vonal közötti távolság.

Következő lépésként a körök középpontjainak elmentése következik. Mindezt úgy teszem meg, hogy közben szűröm is az esetlegesen tévesen felismert köröket. A korábban felismert és eltárolt szimbólumokat egy bounding téglalapban mentettem el egy listában, így csak azokat a köröket mentem el, amelyeknek a középpontja egy adott szimbólumot körülvevő téglalap határain belül van. Ezzel kiszűrtem azokat az esetleges hamis pozitív köröket, amelyek nem valamilyen objektum közelében vannak.



23. ábra: Az objektumoknál felismert körök megrajzolása

Ezek után egy vizsgálatot végzek azokon az objektumokon, amelyeknél detektált kört az algoritmus, hogy az az objektum hangjegy-e. Ezt a keras modell segítségével hajtom végre, ugyanis az már beosztályozta az összes elemet, így egy feltétel vizsgálattal megnézem, hogy olyan osztályba tartozik-e a szimbólum, amelyik egy hangjegynek az osztályba. Ezután már csak ezeket a köröket tárolom el.

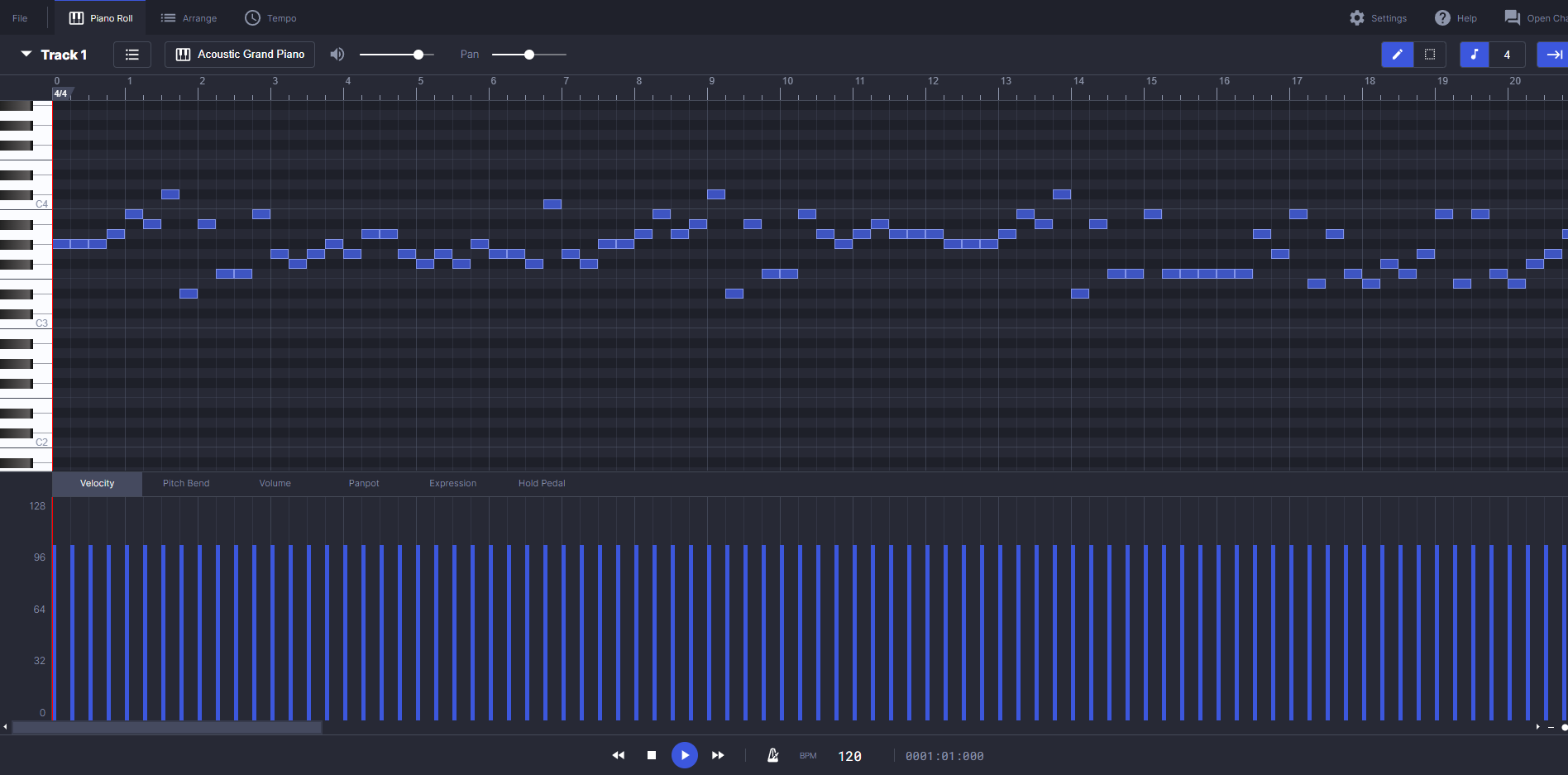
### Hangjegyek pontos elhelyezkedése

Ezután csinálok egy listát, amelybe a végső felhasználásra alkalmas hangjegyek fejeinek koordinátáit és hangmagasságát tárolom le. Ezt a listát feltöltöm annyi üres listával, ahány sor van a kottán, hogy soronként tudjam rendezni, majd végig iterálok az imént elmentett körökön, és megkeresem a hozzájuk legközelebbi vonalat, amelyet a szintén korábban létrehozott listából nyerem ki, amiben a vonalak vannak eltárolva. A fejek elhelyezkedését az alapján határozom meg, hogy vonalon helyezkednek-e el vagy két vonal között, illetve a pontos vonalat vagy vonalközt is megadom. Ez alapján fogja megkapni az adott kör a pitch, azaz a hangmagasság számot, amely 1-11 között van, hiszen 5 vonal van, 4 vonalköz, illetve lehet a legfelső vonal felett és a legalsó alatt is.

Ezután x koordináta szerint sorba rendezem a listát, hogy olyan sorrendben legyenek, ahogyan a kottaolvasás is történik. A kimenet konkrét legenerálása előtti utolsó lépés, hogy egy új listába átmentem az előbbi listából a hangmagasságokat x szerinti sorrendben, de a koordinátákat nem, mert arra nem lesz szükség a midi konvertálásnál.

### MIDI fájl generálása

Most, hogy minden előfeltétel megvalósult, jöhet a program végső lépése, a midi fájl létrehozása a szükséges adatokkal. Ezt a pretty\_midi nevű függvénykönyvtár segítségével hajtom végre, amelyet első lépésként beimportálok. Következő lépésként definiálok egy függvényt, amely a konvertálást hajtja végre. Két bemeneti paramétere van, a path, vagyis annak a helye, ahová a fájlt létrehozza, illetve a notes, vagyis a hangjegyek. A függvényen belül megadom, hogy milyen hangszerrel akarom lejátszani a hangot, valamint, hogy honnan induljon és milyen hosszan szóljon egy-egy hang. Mindezek után meghatározom a hangjegyek lejátszási sebességét és hogy mettől-meddig tart az adott hang, így teszi őket egymás után, valamint a hangmagasságot is hozzárendeli mindegyik hangjegyhez. Az algoritmus utolsó lépése a midi fájl tényleges létrehozása, amit a song.write() függvény hajt végre. Miután mindent beállítottunk a függvényben, meghívom a metódust, megadom neki a létrehozás helyét, illetve azt a listát, amelybe a hangmagasságokat eltároltam a kottaolvasásnak megfelelő sorrendben. Ezen értékek alapján pedig létrejön a midi fájl, amelyet szinte bármilyen hang lejátszó program vagy kottaolvasó szoftver képes értelmezni.



24. ábra:A szoftver által generált midi fájl felépítése

# A program tesztelése

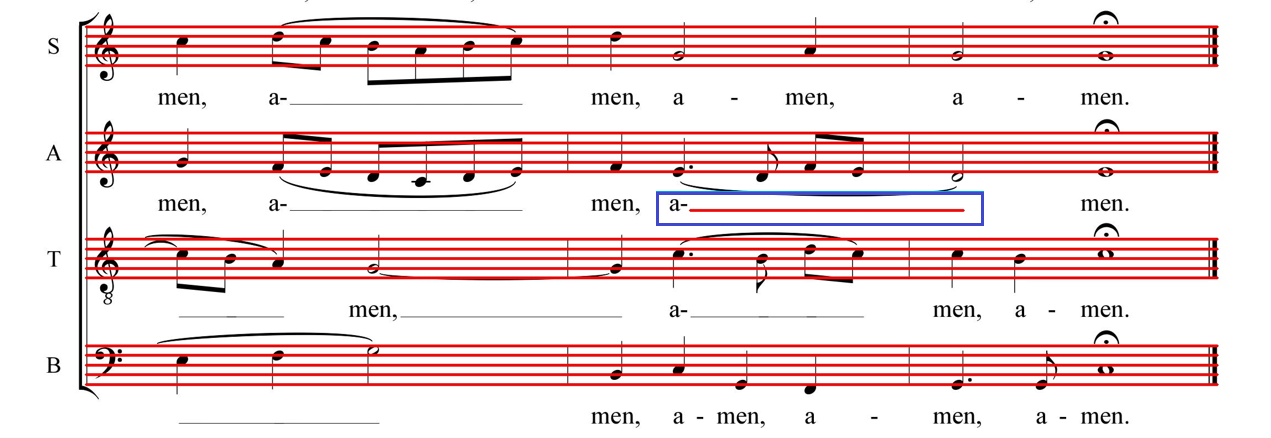
A program elkészítése után nem marad más hátra, mint különböző bemenetekkel tesztelni a működését. A bemeneti képek tartalma és mérete eltérő, a formátumuk pedig JPG. Többféle tesztelési módszer létezik, amelyeket szoftverfejlesztésnél lehet alkalmazni, de a választásom a unit tesztelésre esett. Ezzel a metódussal a programot felépítő kisebb egységeket külön tesztelem. Egy különálló programrészt hívnak unitnak(egységnek), amelyeknek a függvényeit vizsgálom, hogy eltérő bemenetek esetén megfelelő eredményt ad-e vagy hibásat. Azért erre a módszerre esett a választásom, mert a program nagyon egyszerűen izolálható önálló részekre, sőt már a fejlesztés során is külön lettek tesztelve egymástól, illetve ennek a módszernek a használatával a hibák helyének meghatározása roppant egyszerű.

## Vonalak keresése

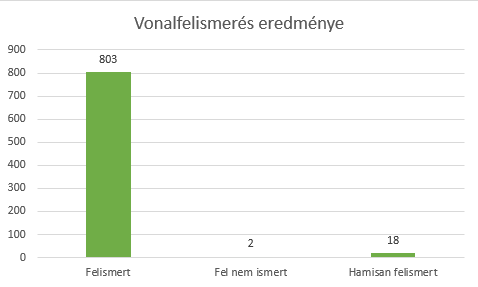
Az első lépés a kottavonalkereső algoritmus tesztelése. A felismert kottavonalakat piros színnel jelöli meg a képen, ez alapján eredménye könnyen ellenőrizhető. A tesztesetek során összesen 805 kottavonal szerepelt a kottákon, amelyeket az algoritmusnak fel kellett ismernie. Ebből összesen 2 darabot nem ismert fel a program, viszont 18 esetben talált olyan vonalat, ami nem kottavonal. Összességében ez a része a programnak sikeresen működőnek mondható.



25. ábra:Példa fel nem ismert kottavonalra



26. ábra: Példa hamisan felismert kottavonalra



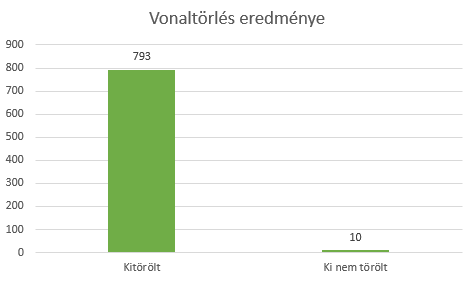
27. ábra: Oszlopdiagram a vonalfelismerés hatékonyságáról

## Vonalak kitörlése

A következő egység, amit tesztelek, az a vonalak kitörlése. A 803 felismert kottavonalból 10 kivételével mindet kitörölte, így ez a programrész is hatékonynak mondható.



28. ábra: Példa ki nem törölt kottavonalakra

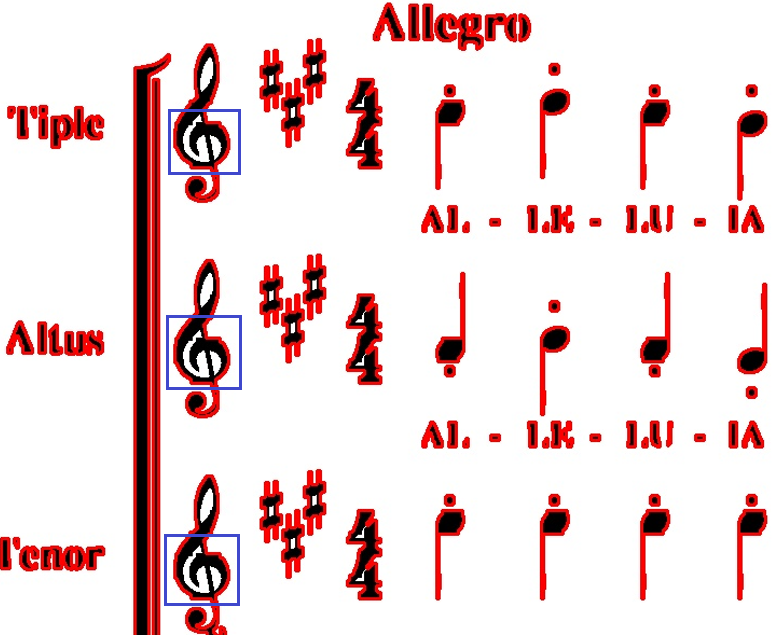


29. ábra: Kottavonalak kitörlésének eredménye

## Kontúrok keresése

A kontúrok keresésének tesztelésénél nem számoltam meg a kottákon található összes komponenst azok számának nagysága miatt. Azonban mivel az algoritmus szintén kiemeli piros színnel a megtalált elemeket, így az egyszerű szabadszemmel történő ellenőrzés során is megtalálhatóak az esetleges hibák. Két hibatípus fordult elő ennél a programegységnél. Az egyik hibatípus a violinkulcsok kontúrjainál van, a másik pedig a korábbi vonaltörlési hibáknál fordul elő, amely nem közvetlenül ennek az egységnek a hibás működését jelenti.

Ettől a két hibától eltekintve a kontúrkeresés tökéletesen működik a kották többi elemén.



30. ábra: Kontúrkeresési hiba a violinkulcsok közepén



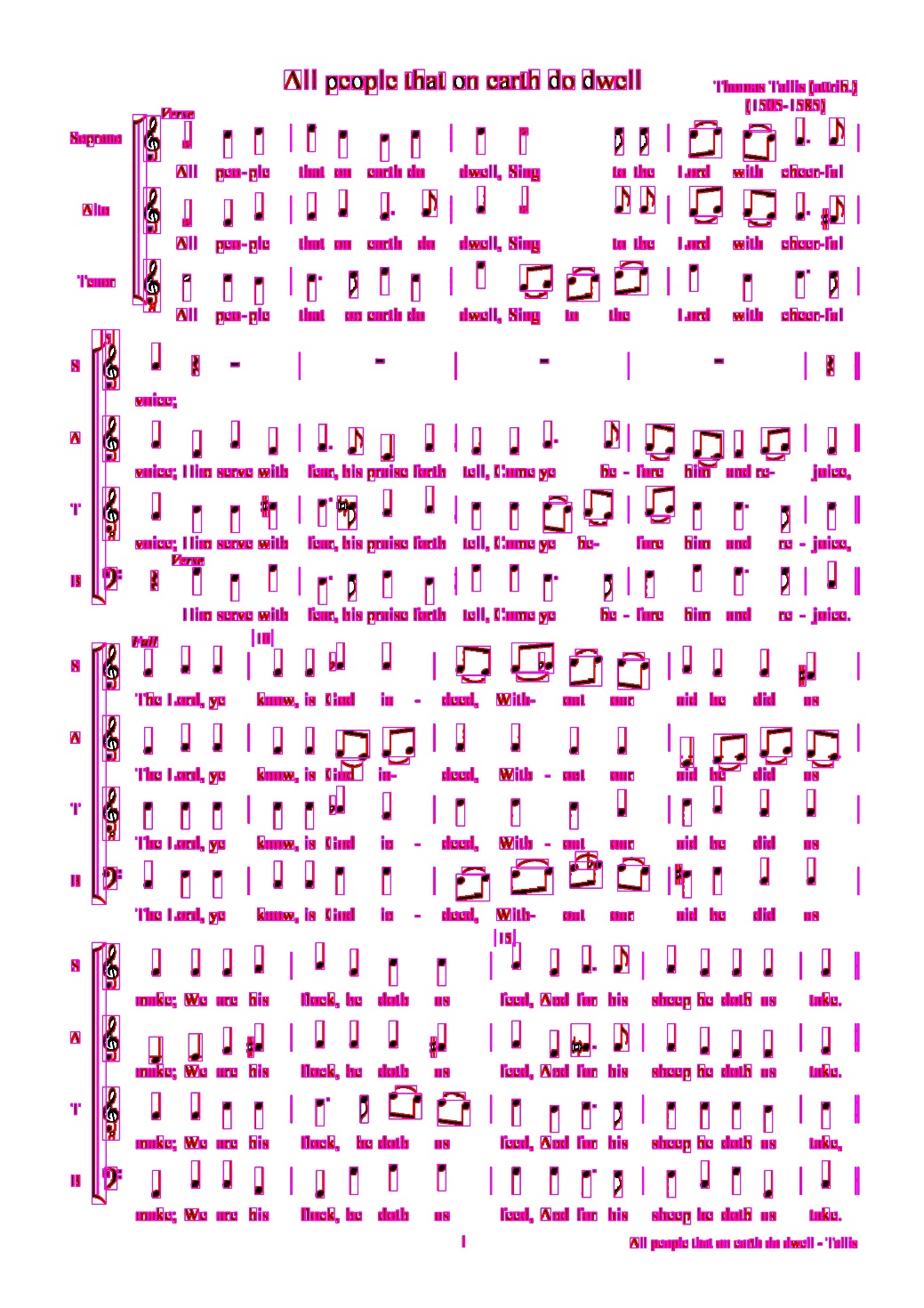
31. ábra: Vonaltörlési hibából adódó kontúrkeresési hiba



32. ábra: Kontúrkeresés átlagos eredménye egy kottán

## Alakzatok eltárolása

Ennél a programrésznél is a manuális végig ellenőrzés módszerét alkalmaztam, a túl nagy mennyiségű objektumok miatt, azonban mint az az alábbi ábrán is látható, az alakzatok körbekerítése, majd eltárolása sikeresen végbemegy mindegyik tesztkép esetében.

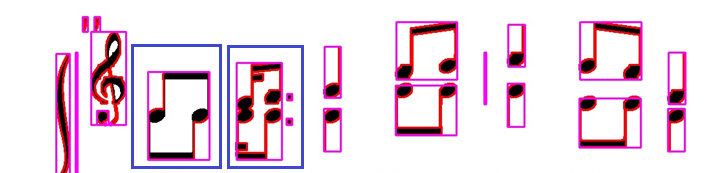


33. ábra: Alakzatok körbe rajzolásának átlagos eredménye

Kettő esetben nem sikerült az egységnek sikeresen ellátni a feladatát, egyik ilyen eset a korábbi hibákból származó ki nem törölt vonalak mellet és rajta lévő alakzatok bekerítése, másik eset pedig két hangjegy fej összeérésekor a hangjegyeket egy hangjegyként kezeli. Ezeknek az előfordulása azonban nem számottevő, így ez a programrész is megfelelően működőnek mondható.



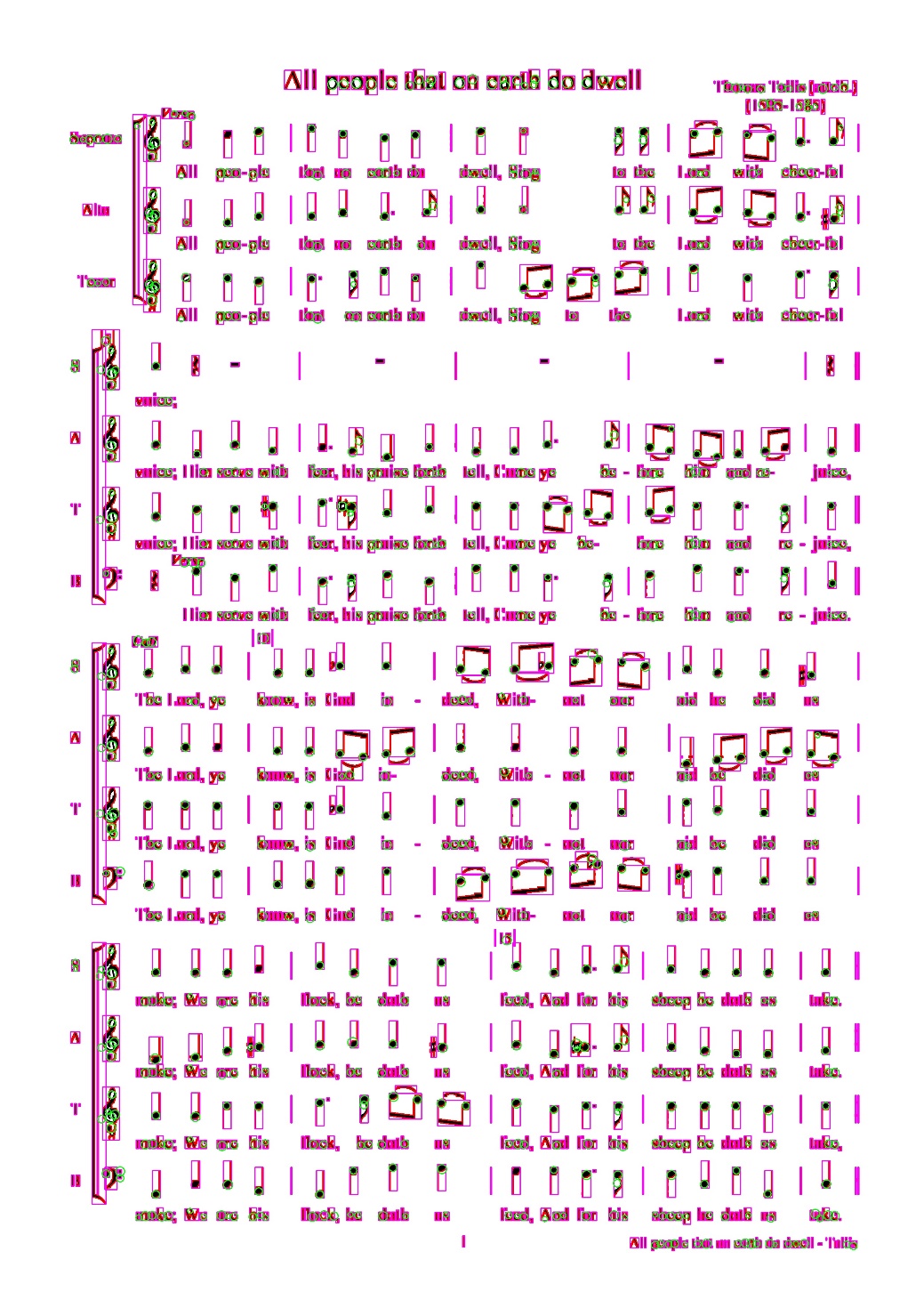
34. ábra: Korábbi hibákból adódó hibás működés



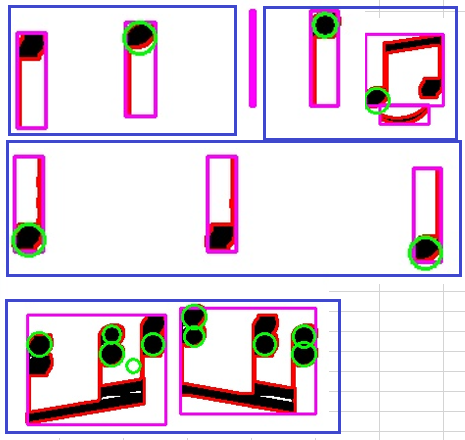
35. ábra: Összeérő hangjegyeknél történő körbe kerítési hiba

## Körök detektálása

A következő része a szoftvernek a kottán található hangjegy fejek megtalálása, kör alakzatok keresésével. A tesztelt bemenetekre mind hasonló eredményt adott ez a funkció is, a hangjegyek túlnyomó többségén detektálta a fejeket. Konkrét számadatokkal itt sem tudok dolgozni, hiszen habár az eltárolt szimbólumok számát itt már le tudom kérdezni a programból, annak egy jelentős része olyan alakzat, ami nem hangjegy. Illetve ennél a program egységnél sok olyan kört is észlel az algoritmus, amely hamis pozitív, azaz tévesen érzékeli körnek, így hatékonysági statisztikát ezekből az adatokból nem lehet készíteni. Mivel a következő lépésben ezeket az eseteket kiszűrjük, így ez nem veszélyezteti a szoftver helyes működését. Detektálási hibák ebben a részében is előfordulnak a programnak, azonban itt konkrét tendenciát már nem lehet nyomon követni. Nincsenek állandó hibaesetek, inkább véletlenszerűek, amelyeknek okát nem sikerült megfejteni. Mindezek mellett az algoritmus a hangjegy fejek szinte összes előfordulását megtalálja. Az alábbi ábrán is látszik, hogy mindössze néhány esetben nem tudja csak detektálni a fejeket az algoritmus.



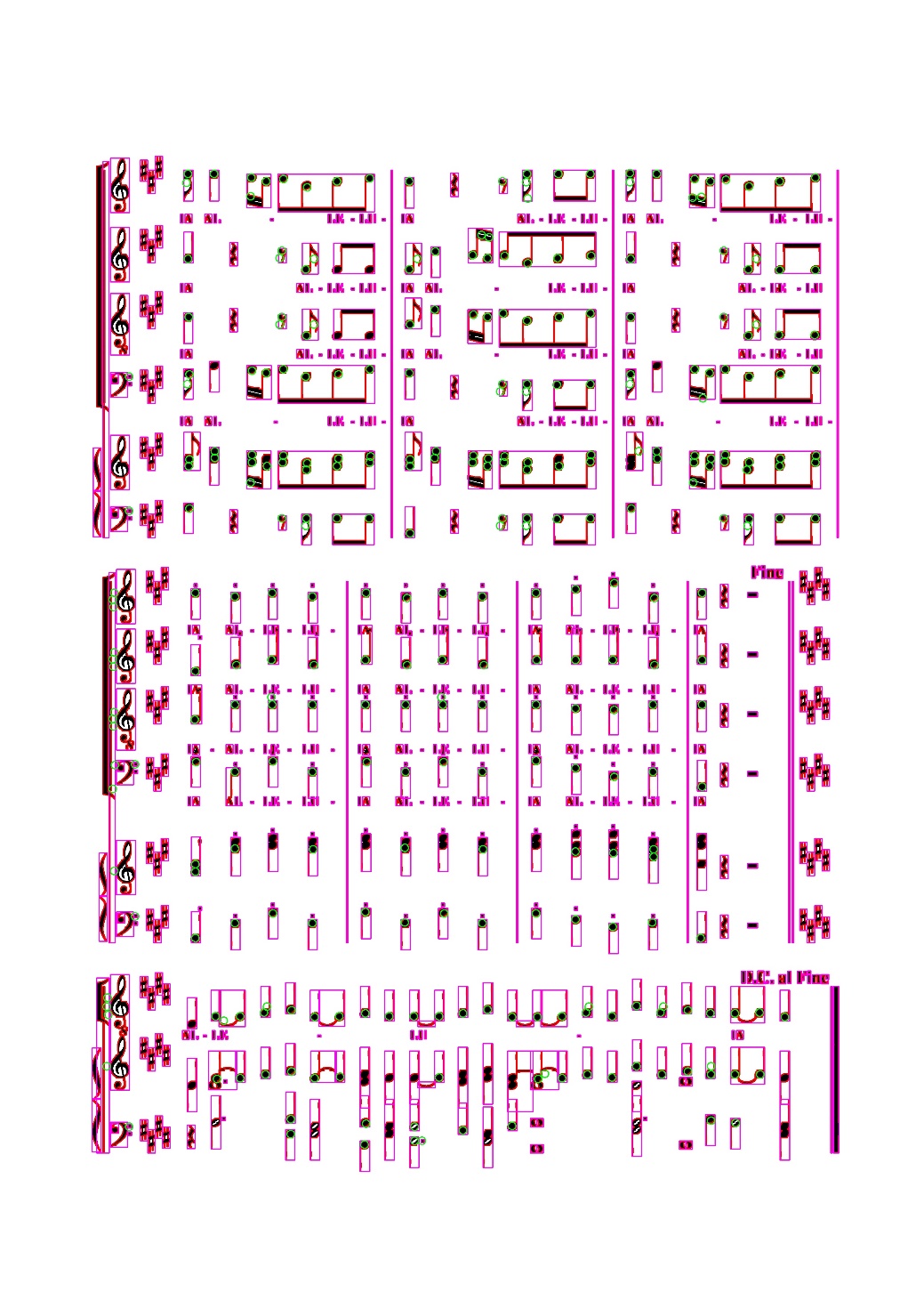
36. ábra: Hangjegy fejek keresésének átlagos eredménye



37. ábra: Néhány példa hangjegy fej detektálási hibára

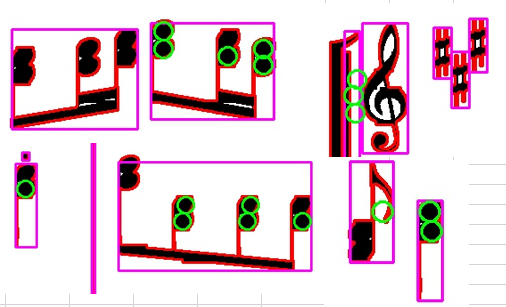
## Körök szűrése

A következő lépés a megtalált körökből kiszűrni azokat, amelyek hangjegyekhez tartoznak, vagyis nagy valószínűséggel valóban hangjegy fejek. Ennek a megvalósításához az általunk felépített neurális modellt alkalmazzuk, így ebben a részben annak a működését teszteljük. Összességében a tesztek eredményéről elmondható, hogy a nem hangjegyekhez tartozó köröket egészen jó arányban kiszűri, azonban vannak tévesen bent maradó körök is. Leginkább a violinkulcsoknál fordul ez elő, de egy-két „szün” jel esetében is előfordult ez a probléma. Ennél gyakoribb probléma azonban az, hogy a szekvenciális modell által történt csoportosítás nem mindig hangjegyként azonosított tényleges hangjegyeket, így akadnak olyan elemek, amelyek bár hangjegyek, de mégis kiszűrte azok fejét a program. Ez főként a duplafejű hangjegyeknél és az összeérő hangjegyeknél fordul elő, de nem mindegyik esetben.



38. ábra: Hangjegyek fejeinek megszűrésének átlagos eredménye

Mint az ábrán látható, maradnak bent nem hangjegyhez tartozó körök, illetve kerülnek ki olyanok, amelyeknek ideális esetben nem kéne. Ennek a hatékonyságán a keras modell rétegeinek esetleges átalakításával vagy finomításával lehet javítani.



39. ábra: Példák hibásan bent maradt és tévesen kiszűrt körökre

## Kimenet

A legenerált kimenetek tesztelésekor két fő szempontot vizsgáltam. Az egyik, hogy a generált fájlok valóban lejátszható és hallható hangfájlok legyenek, illetve, hogy a hangjegyek fejei, amelyeket a fájl generálásához használt, a megfelelő hangmagasságban kerültek-e bele a midi fájlba. Ezek ellenőrzésére egy ingyenesen elérhető internetes hangfájl lejátszó és megjelenítő alkalmazást használtam. A kimeneti fájlok hangjával és lejátszhatóságával egyik teszt során sem volt gond, mindegyik esetben tökéletesen működött. A hangmagasságok ellenőrzéséhez összevetettem a hangjegyek elhelyezkedését a kottán és a midi fájlban. Néhány esetben megfigyelhető, hogy a közvetlen egymás alatt lévő hangokat (egyik a kottavonalon, másik az alatta/felette lévő vonalközben) ugyanannak a hangmagasságnak érzékeli, így azok nem tökéletesen kerülnek bele a midi fájlba. Ezen kívül még a tévesen bent hagyott körök is bekerülnek, szóval a végkimenetel nem minden esetben a teljes tükörképe a bementként kapott kottának.



40. ábra: Hangmagasság eltérés a bemenet és a kimenet között

# Összegzés

A dokumentációban részletesen megfogalmaztam, hogy hogyan készítettem el egy zenei kotta digitalizálására alkalmas szoftvert. Ehhez minden szükséges elméleti információt leírtam, valamint a fejlesztést is lépésről lépésre bemutattam. A kész program egy képet kap egy zenei kottáról, amit különböző műveltek segítségével midi kiterjesztésű fájlra konvertál. Ennek megvalósításához különböző képfeldolgozási funkciókat és mesterséges intelligenciát használtam.

A program részegységei és azok együttműködése is be lett mutatva a dokumentációban, illetve tesztelve is lett. A tesztelés során bíztató eredményeket kaptam, azonban van még fejlesztési potenciál a szoftverben. Főként a keras modell finomításával lehetne a program hatékonyságát növelni, de az egyes képfeldolgozó algoritmusok paramétereinek megváltoztatásával is érhetek el akár ennél is jobb eredményeket.

Az elkészült szoftver egyik fő erősségének érzem a rugalmasságát és a konfigurálhatóságát. A fejlesztés során rengeteg új ismerettel bővültem, főként a mesterséges intelligencia és a gépi látás területein. Az OpenCV-ről is átfogó ismereteket szereztem, valamint a Python programozási nyelvben is elmélyültem a szoftver készítése közben. Ezek mind olyan tapasztalatok és ismeretek, amelyek nagyon hasznosak lesznek számomra, ha a későbbiekben is szoftverfejlesztéssel fogok foglalkozni.

# Irodalomjegyzék

1. Martín Abadi: TensorFlow: learning functions at scale. Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, 2016
2. Gary Bradski, Adrian Kaehler: *Learning OpenCV: Computer Vision with the OpenCV Library,* 2008, pp. 153-158.
3. John Canny: A Computational Approach to Edge Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1986, pp. 679–698.
4. Csatlós István: *Programozási nyelvek kialakulása, fejlődése,* 2014, pp. 1-4.
5. Dagao Duan, Meng Xie, Qian Mo, Zhongming Han, Yueliang Wan: *An improved Hough Transform for line detection,* International Conference on Computer Application and System Modelling, 2010, pp. 354-355.
6. Dumitru Erhan, Christian Szegedy, Alexander Toshev, Dragomir Anguelov, *Scalable Object Detection using Deep Neural Networks,* 2013, pp. 1-3.
7. C. Galambos, J. Matas, J. Kittler: *Progressive Probabilistic Hough Transform for line detection,* Computer Society Conference on Computer Vision and Pattern Recognition, 1999, pp. 554-556.
8. Reagan L. Galvez, Argel A. Bandala, Elmer P. Dadios, Ryan Rhay P. Vicerra, Jose Martin Z. Maningo, *Object Detection Using Convolutional Neural Networks,* TENCON Region 10 Conference, 2018, pp. 2023-2024.
9. Dr. Loványi István: Morfológiai képfeldolgozás, 2014, pp. 1-56.
10. Ramesh Jain, Rangachar Kasturi, Brian G. Schunck: *Machine vision,* 1995, pp. 1-31.
11. Prateek Joshi, David Millan Escriva, Vinicius Godoy: *OpenCV By Example,* 2016, pp. 5-10.
12. Kondorosi Károly, Szirmay-Kalos László, László Zoltán: *Objektum orientált szoftverfejlesztés,* 2007, pp. 53-62.
13. *Nikhil Ketkar: Introduction to Keras. Deep Learning with Python,2017, pp. 97–111.*
14. Navin Kumar Manaswi: Understanding and Working with Keras. Deep Learning with Applications Using Python, 2018, pp. 31–43.
15. Maurício Marengoni, Denise Stiringhini: *High Level Computer Vision Using OpenCV,* SIBGRAPI Conference on Graphics, Patterns, ands Images Tutorials, 2011, p. 13
16. Alexander Mordvinstev, Abid K., *OpenCV-Python Tutorials Documentation,* 2017, pp. 87-106.
17. Maosud Nosrati: *Python: An appropriate language for real world programming,* World Applied Programming, Vol. 1 No. 2, 2011, pp. 110-114.
18. Simon Just Kjeldgaard Pedersen: Circular Hough Transform, Aalborg University, Vision, Graphics, and Interactive Systems, November 2007, pp. 1-4.
19. Colin Raffel, Daniel P. W. Ellis. “Intuitive Analysis, Creation and Manipulation of MIDI Data With pretty\_midi”, 15th International Society for Music Information Retrieval Conference, 2014, pp. 1-2.
20. Xu, Z., Baojie, X., & Guoxin, W.: Canny edge detection based on Open CV., 13th IEEE International Conference on Electronic Measurement & Instruments (ICEMI)., 2017, pp. 53.
21. Wang Zhiqiang, Liu Jun, *A Review of Object Detection BAsed on Convolutional Neural Network,* Chinese Control Conference, 2017, pp.11104-11108.
22. https://docs.python.org/3.0/whatsnew/3.0.html (megtekintve: 2021. április 16.)