Gépi látás dokumentáció

Kártya detektáló algoritmus

Kenderesi Antal László

Mérnökinformatikus BSc / BA

2022/23/1

Tartalomjegyzék

1. Bevezetés

Féléves feladatomnak egy kártyafelismerő alkalmazás megvalósítását választottam. Az alkalmazás célja, hogy egy bemenetként kapott képen meghatározza az előre megadott kártyapaklihoz tartozó kártyalapok pozícióját és típusát, majd ezeket az információkat megjelenítse a kimeneti képen. A feladat megoldásához a hagyományos francia kártyát tartottam megfelelőnek, mivel ez az egyik legelterjedtebb játékkártyatípus.

A francia kártya olyan kártyajátékokban használatos, mint póker, bridge, és pasziánsz. A pakli minden kártyája lekerekített téglalap alakú, és rendelkezik egy rang, valamint egy szín értékkel. Összesen a pakli 52 laptól áll, ami 13 különböző rangú kártyát tartalmaz 4 féle színben. A rang értékek lehetnek számok 2-től 10-ig, bubi (J), dáma (Q), király (K), és ász (A). Ezek 4 színváltozata a fekete pikk, a piros káró, a fekete treff és a piros szív vagy kőr. A lapokat megkülönböztető szimbólumok a kártya 2 átellenes sarkában találhatók. Emellett a kártyák közepén egyéb grafikai elemek is előfordulnak ezzel meg inkább egyedivé és megkülönböztethetővé téve mind az 52 kártyalapot. A francia kártyák pontos megjelenése függ a gyártótól és a design tematikától. A legtöbb kártyapakli további egyéb kártyákat is tartalmazhat, mint 2 darab joker és reklám kártyák.

A francia kártyáknak számos vizuális tulajdonsága van, amelyeket ki lehet használni egy kártya detektáló algoritmus tervezéséhez:

1. A kártyák téglalap alakú formája, amit képfeldolgozó technikákkal lehet kihasználni, például éldetektálással és kontúrelemzéssel.
2. Az egyes kártyák konkrét rangja és színe, amely a kártyák sarkain lévő számok és szimbólumok, valamint a kártyák előlapján lévő grafikák és minták elemzésével észlelhető.
3. Az kártyák színe, amely színalapú képszegmentációs technikákkal detektálható.

Általánosságban elmondható, hogy a kártya detektáló algoritmusnak képesnek kell lennie arra, hogy azonosítsa és kinyerje ezeket a vizuális információkat egy kártyát tartalmazó képből, hogy pontosan meghatározhassa a kártya rangját, színét és egyéb releváns információkat. Ezt az algoritmust képfeldolgozási technikák és gépi tanulási módszerek kombinációjával célszerű megvalósítani.

2. Elméleti háttér

2.1. OpenCV könyvtár

2.1.1. cvtColor

Az OpenCV könyvtár *cvtColor* függvénye a kép egyik színtérből a másikba konvertálására szolgál [1]. A függvény bemeneteként az eredeti képet, a kívánt kimeneti színteret és az átalakítási folyamatot vezérlő opcionális paramétereket kell megadni. Ezután a függvény a konvertált képet adja vissza kimenetként.

A feladat során ezt a függvényt kizárólag szürkeárnyalatos konverzióra használtam. Ennek a konverziónak több módja közül választhatunk a függvény használata során. A függvény alapértelmezés szerint a fényerősség nevű módszert használja. Ez a módszer a következő képlet alapján számítja ki az egyes képpontok szürkeárnyalatos értékét: 0,21 * R + 0,72 * G + 0,07 * B, ahol R, G és B a pixel piros, zöld és kék értékei. A módszer nagyobb súlyt ad a zöld csatornának, amely a legérzékenyebb az emberi szemre. Gyakran használják olyan képfeldolgozási feladatoknál, amelyek jó vizuális minőséget igényelnek a kimeneti képben.

2.1.2. inRange

Az OpenCV könyvtár *inRange* függvénye a kép küszöbértékének kiszámítására szolgál [1]. A küszöbérték számítás egy általános képfeldolgozási művelet, amelyet gyakran használnak a képfeldolgozás során a kép különböző területeinek szegmentálására vagy akár zaj eltávolítására. Az eljárás során egy szürkeárnyalatos vagy színes képből bináris kép készül. A bináris képen minden pixel fekete vagy fehér, attól függően, hogy elér-e egy bizonyos küszöbértéket. Az *inRange* függvény bemenetként egy képet, egy alsó küszöbértéket és egy felső küszöbértéket szükséges megadni. Ezután a következő logikát alkalmazza a kép minden pixelére:

- Ha a pixelérték kisebb, mint az alsó küszöbérték, a függvény a pixelértéket 0-ra (feketére) állítja.
- Ha a pixelérték nagyobb, mint a felső küszöbérték, a függvény a pixelértéket 255-re (fehérre) állítja.
- Ha a pixelérték az alsó és a felső küszöbérték között van, a függvény a pixelértéket változatlanul hagyja.

2.1.3. findContours

Az OpenCV könyvtár *findContours* függvénye kinyeri a képen lévő objektumok körvonalait [1]. Bemenetként egy képet igényel, ami alapján visszaadja a kontúrok vagy körvonalak listáját, amelyek a képen lévő objektumok határait reprezentálják. Ezek a körvonalak ezután különféle célokra használhatók, például objektumfelismerésre, tárgykövetésre vagy képszegmentálásra.

A *findContours* függvény meghívásakor, a függvény több képfeldolgozási műveletet hajt végre a bemeneti képen, hogy megkeresse a képen lévő objektumok körvonalait. Ezek a műveletek a következők:

1. Küszöbérték számítás: A *findContours* függvény először egy küszöbérték-műveletet alkalmaz a bemeneti képre, amely a képet bináris képpé alakítja, ahol minden képpont fekete vagy fehér. Ez a művelet a kép egyszerűsítésére és a képen lévő objektumok kontúrjainak könnyebb megtalálására szolgál.
2. Kontúrfelismerés: A küszöbérték meghatározása után a függvény egy kontúrészlelő algoritmust használ a bináris képen lévő objektumok körvonalainak megkeresésére. Ez az algoritmus jellemzően eróziós és dilatációs műveleteket alkalmaz a zaj eltávolítására és a képen lévő objektumok körvonalainak kisimítására.
3. Kontúrközelítés: Miután a függvény a képen lévő objektumok körvonalait észlelte, egy kontúrközelítő algoritmust alkalmaz, ami körvonalakat egyszerűbb alakzatokkal közelíti. Ez azért történik, hogy csökkentse a pontok számát.

2.1.4. contourArea

Az OpenCV könyvtárban található *contourArea* függvény egy körvonal területének kiszámítására szolgál [1]. Ez a függvény bemenetként egy körvonalat igényel, vagyis olyan képpontok listáját, amelyek egy objektum határát jelentik meg a képen. A függvény kimenete a körvonal területe, amit a körvonal alakjától függően különböző matematikai műveletekkel becsül meg.

2.1.5. arcLength

Az OpenCV könyvtár *arcLength* függvénye egy körvonal hosszának kiszámítására szolgál [1]. Bemenetként egy körvonalat igényel, kimenetként a kontúr hosszát adja vissza. A függvény különböző matematikai képleteket használ a körvonal hosszának kiszámításához, a körvonal alakjától függően. Például egy egyenes hossza egyszerűen a végpontjai közötti távolság, míg a görbe hosszát a görbét alkotó egyes szakaszok hosszának összegzésével számítja ki. Használata hasznos a képen lévő objektumok méretének vagy alakjának kiszámításához.

2.1.6. approxPolyDP

Az OpenCV könyvtárban található *approxPolyDP* függvény egy körvonal egyszerűbb alakkal való közelítésére szolgál [1]. Ez a függvény bemenetként egy körvonalat igényel, és egy tűrésértéket, amely a közelítés szintjét szabályozza. Ezután a körvonalat megközelítő sarokpontok listáját adja vissza. A függvény alapértelmezés szerint a Ramer-Douglas-Peucker algoritmust használja. Ez az algoritmus egy sor egyenes vonalat illeszt a bemeneti körvonalhoz, és eltávolítja azokat a pontokat,

amelyek távolabb vannak az illesztett vonalaktól, mint a tűrésérték. Az algoritmus gyors és hatékony, de előfordulhat, hogy nem mindig a legjobb közelítést adja.

2.1.7. getPerspectiveTransform

Az OpenCV könyvtár *getPerspectiveTransform* függvénye egy kép perspektivikus transzformációjának kiszámítására szolgál [1]. A függvény bemenetként négy képpontot igényel az eredeti képen és négy képpontot a kimeneti képen. A függvény kimenete a kiszámított perspektíva-transzformáció mátrix. A perspektíva-transzformáció mátrix segítségével egy képet egyik perspektívából a másikba transzformálhatunk. A *getPerspectiveTransform* függvény a perspektíva-transzformáció mátrixot úgy számítja ki, hogy a bemeneti és kimeneti pontok alapján elsőfokú egyenleteket old meg. A perspektíva-transzformáció mátrixot kiszámítása után felhasználhatjuk a kép átalakítására az OpenCV könyvtár *warpPerspective* függvényével.

2.1.8. warpPerspective

Az OpenCV könyvtár *warpPerspective* függvénye perspektíva transzformációt végez egy képen [1]. A függvény bemenetként egy képet, egy perspektíva-transzformáció mátrixot és a kimeneti kép méretét igényli, majd az átalakított képet adja vissza kimenetként. A perspektíva-transzformáció mátrixot a *getPerspectiveTransform* függvény segítségével számítjuk ki. A *warpPerspective* függvény ezt a mátrixot használja arra, hogy a bemeneti képet a kívánt perspektívára alakítsa át úgy, hogy a bemeneti kép minden egyes pixelét leképezi a kimeneti kép új helyére. A *getPerspectiveTransform* és *warpPerspective* függvények számos képfeldolgozási feladathoz hasznosak, mint például panorámakép összefűzés vagy objektumfelismerés.

2.2. TensorFlow keretrendszer

2.2.1. Sequential

A TensorFlow könyvtárban található *Sequential* modellt azért nevezik szekvenciálisnak, mivel a neurális hálózat rétegeit egymást követően határozza meg, és az adatok a rétegek definiálásának sorrendjében haladnak át a hálózaton [2]. A szekvenciális modell a TensorFlow legegyszerűbb és legelterjedtebb modellje, amely kiválóan alkalmas egyszerű, lineáris adatáramlású neurális hálózatok létrehozására.

2.2.2. Conv2D

A TensorFlow keretrendszer *Conv2D* rétege egy konvolúciós réteg, amelyet általában a konvolúciós neurális hálózatokban (CNN) használnak képosztályozásra és egyéb feladatokra [2]. Ez a réteg szűrőkészletet alkalmaz a bemeneti adatokra, és minden szűrő egy feature map-et hoz

létre, amely a bemeneti adatok más-más aspektusát emeli ki. A réteg legfontosabb paraméterei a következők:

- filters: A rétegben használandó szűrők, vagyis konvolúciós kernelek száma. A réteg egyszerre több szűrőt hoz létre, amiket csúszó-ablakos módszerrel külön-külön iterál végig az eredeti képen. Ezt követően egy *Tensor* objektumban tárolja minden szűrő eredményét.
- kernel_size: A rétegben használandó szűrők mérete. Ezt általában két egész számmal adják meg, amelyek a szűrők magasságát és szélességét jelentik pixelben.
- activation: A szűrők alkalmazása után a létrejött feature map-eken további műveleteket hajthatunk végre hasonlóan a hagyományos teljesen összekapcsolt neuronrétegekhez. Ez esetben az aktivációs függvényt külön alkalmazzuk minden feature map összes pontjára. Például a ReLU aktivációs függvény megadásával a szűrők minden nem negatív eredményét 0-ra állíthatjuk.

2.2.3. MaxPooling2D

A TensorFlow keretrendszer *MaxPooling2D* rétege arra hivatott, hogy a réteget megelőző *Conv2D* réteg kimenetét lejjebb skálázza [2]. A konvolúciós réteghez hasonlóan itt is meg kell adnunk a mintavételezési ablak méretét egész számokkal. A réteg a bemenetként kapott feature map-eken végig iterálva minden alkalommal a mintavételi ablak legnagyobb értékét veszi kimenetként. Ennek a műveletnek két fontos hatása is van a hálózat működésére nézve. Egyrészt redukálja a feature map-ek méretét, vagyis a későbbi rétegekben kevesebb számítást kell emiatt végezni. Másrészt kiválogatja a feature map-ek legfontosabb pontjait, mivel általánosságban a szűrők magas intenzitású kimenetei fontos tulajdonságokra utalnak.

2.2.4. Flatten

A TensorFlow keretrendszer *Flatten* rétege egy olyan réteg a neurális hálózatban, amely a bemeneti *Tensor* -t kétdimenziós *Tensor*-rá lapítja, ahol az új méreteket a batch méret és az összes feature pont határozza meg [2]. Gyakran használják előfeldolgozási lépésként, mielőtt átadnák az adatokat egy teljesen összekapcsolt rétegnek, amely kétdimenziós *Tensor* objektumot vár bemenetként.

2.2.5. Dense

A TensorFlow keretrendszer *Dense* rétege egy teljesen összekapcsolt neuronréteget jelent egy neurális hálózatban [2]. A teljesen összekapcsolt rétegben minden neuron kapcsolódik az előző és a következő rétegek minden neuronjához. Ez teszi lehetővé, hogy a hálózat megtanuljon bonyolult nemlineáris kapcsolatokat a bemenet és a kimenet között.

3. Alkalmazás megvalósítása

3.1. Tervezés

3.1.1. Használt technológiák

Az alkalmazás megvalósításához a **Python** programnyelvet választottam, azon belül is a projekt készítésének idején legfrissebb 3.10.8-as verziót. A Python az utóbbi évek egyik legnépszerűbb programnyelve. Népszerűségének fő oka a könnyen tanulható és használható szintaxis. Emellett rengeteg gépi látás és gépi tanulás tematikájú könyvár és keretrendszer érhető el a nyelvhez, ezzel egyszerűbbé téve összetettebb alkalmazások fejlesztését. A projektet vegyesen alkotják Python scriptek és Jupyter Notebook fájlok. A **Jupyter Notebook** egy webalapú platform, amely kód futtatásához biztosít interaktív grafikus felületet. Használata a hagyományos scriptekkel szemben olyan kényelmi funkciókat nyújt, mint képek beágyazott megjelenítése, kódblokkok futtatása egymástól függetlenül, illetve a kódblokkok futási idejének kijelzése.

A Python nyelvhez készített könyvtárakat és keretrendszereket a pip csomagkezelő rendszerrel telepíthetjük. A projekthez használt csomagok listája a requirements.txt fájlban található. A listában szereplő első csomag a NumPy. A **NumPy** egy nyílt forráskódú könyvtár Python-hoz, amely számos bonyolult matematikai műveletet megvalósító függvényt, és számábrázoláshoz használható adattípus osztályt tartalmaz. Hatékonyágából adódóan szinte minden projektben megtalálható, beleértve más programkönyvtárakat is. A projekt során az alkalmazás képfeldolgozással kapcsolatos funkciói az OpenCV könyvtár segítségével jöttek létre. Az **OpenCV** egy széles körben elterjedt nyílt forráskódú gépi látás témájú programkönyvtár, C++, Python, Java és MATLAB támogatással. Az alkalmazás neurális hálózat komponensei a TensorFlow gépi tanulás keretrendszerrel valósultak meg. A **TensorFlow** keretrendszer szintén nyílt forráskódú, valamint több programnyelvhez érhető el, melyek közül leginkább támogatott a Python. Végezetül a Jupyter Notebook-okban különböző képek megjelenítésére a Matplotlib könyvtárat használtam. A **Matplotlib** adatok vizualizálására és diagramok készítésére alkalmas.

A fejlesztő környezetként a **Visual Studio Code** alkalmazást használtam, mivel támogatja a notebook fájlok megnyitását is.

3.1.2. Algoritmus működésének terve

A kártyadetektáló algoritmus 5 egymást követő lépésből áll:

1. Beolvasás
2. Szegmentáció
3. Transzformáció
4. Osztályozás
5. Kimenet

Az első lépésben beolvassuk a bemeneti képet. A kép 3 színcsatornával rendelkezik, felbontása és képaránya tetszőleges. A második lépésben elkülönítjük a képen található kártyákat a környezettől. Ezt követően az egyes kártya példányokat is elkülönítjük egymástól, így egy képen több kártya detektálására is lehetőség nyílik. A harmadik lépésben előre meghatározott felbontásra és perspektívára transzformáljuk a kártya példányokat. A lépés lényege, hogy minden kártyát egységes részletességgel, valamint egységes nézőpontból vizsgáljunk. A transzformáció eredményeképpen a kártyák megkülönböztető jelei mindig azonos területre esnek a feldolgozott képen, melyek kivágása megalapozza az algoritmus következő lépését. A negyedik lépésben osztályozzuk az előző lépésből származó képkivágásokat. Az osztályozási feladatot neurális hálózatok végzik, amelyek használatához minden körülmény ideális. Az utolsó, ötödik lépésben megjelenítjük az eredményeket az eredeti bemenő kép segítségével. Az eredmények magába foglalják a kártyákhoz tartozó képpontoknak és a kártyák típusának meghatározását.

3.1.3. Megkötések

Az algoritmus kivitelezhetőségét lehetővé téve szükséges néhány megkötést tennünk a bemeneti képekkel kapcsolatban. A megkötésekből származó előnyök elsősorban arra vonatkoznak, hogy az alkalmazás képes lesz elfogadható időn belül, elfogadható pontosságú eredményeket produkálni. A megkötések hátrányai azonban akként jelentkeznek, hogy az alkalmazás nem minden körülmény esetén lesz képes detektálni a kártyákat. Ebből adódóan ügyelnünk kell arra, hogy ezeket a megszorításokat oly módon határozzuk meg, hogy az alkalmazás használhatósága ne csökkenjen jelentősen.

Első lépésként össze kell szednünk minél több lehetséges szituációt, amiket ideális esetben kezelnie kell az alkalmazásnak. Ezt követően meg kell határoznunk egy fontossági sorrendet a szituációk kezelésére vonatkozóan, ami alapján döntést hozhatunk a különböző megkötésekről. Az alábbi felsorolásban találhatóak azok a szituációk, amiket az elkészült alkalmazás nem képes kezelni, valamint javaslatok a szituációk elkerülésére:

- Túlzottan világos háttér a kártya alapszínéhez képest - használjuk hagyományos fehér alapú kártyát és jól elkülöníthető hátteret
- A kártya egy része kilóg a képből, vagy más okokból nem látható - ügyeljünk arra, hogy a fénykép témája teljes egészében benne legyen a képben
- Összeérnek a kártyák - ügyeljünk a kártyák elhelyezkedésére a képen belül
- A kártya nem tölti ki a kép jelentős részét, túl kicsi - ügyeljünk a kamera távolságára a fénykép témájától
- Egyéb zaj körülmények: árnyék vetül a kártyára, túlzott elmosódás - ügyeljünk a fényviszonyokra és a kamera stabilizációjára

Ezzel szemben a következő felsorolás mindazon szituációkat tartalmazza, amiket az alkalmazás kezelni képes:

- Tetszőleges bemeneti képfelbontás és képarány
- Tetszőleges számú kártya a képen
- Tetszőleges kártya pozíció a képen: elforgatás, távolság, dőlésszög

3.2. Implementáció

3.2.1. Alkalmazás felépítése

Az alkalmazás felépítése a következőképpen néz ki:

- **data**: Adatok gyűjtésével és feldolgozásával kapcsolatos scriptek mappája.
    - ***record.py***: A számítógéphez csatlakoztatott webkamera segítségével hoz létre képeket a modellek tanításához és teszteléséhez. Ezek a képek módosítatlan formában, 3 színcsatornával kerülnek mentésre jpg kiterjesztéssel. Mivel több tízezer ilyen kép készült, így nem mindegyik szerepel a GitHub repository-ban helytakarékossági okokból.
    - ***process.ipynb***: A record.py script által generált képeket dolgozza fel. A feldolgozott képeket NumPy array formában a háttértárra menti későbbi felhasználáshoz. A feldolgozott képeket tartalmazó fájlok gyakran meghaladják a GitHub maximum megengedett fájlméretét, ami 50 MB.
- **models**: Neurális hálózat modellek mappája.
    - **rank/*rank_model.ipynb***:
    - **suit/*suit_model.ipynb***:
- **src**: Képfeldolgozó függvények és a kártyadetektáló algoritmus forráskódjainak mappája.
    - ***image_manipulation.py***: Különböző képmanipulációs és képfeldolgozó függvényeket tartalmazó fájl.
    - ***playing_card_detection.py***: A kártyadetektáló algoritmust tartalmazó fájl.
- **test**: A teszteléshez használt képek mappája.
    - **in**: Feldolgozatlan képek mappája.
    - **out**: Feldolgozott képek mappája.
- ***cam_app.py***: A script számítógéphez csatlakoztatott webkamera segítségével olvas be képeket. Ezeket továbbadja a kártyadetektáló algoritmusnak, amelynek kimenete megjelenik a képernyőn.
- ***img_app.py***: Célja megegyezik a cam_app.py fájléval. A különbséget az jelenti, hogy a háttértárról olvas be képeket. A feldolgozott képeket opcionálisan megjeleníti a képernyőn, vagy a háttértárra menti.
- ***README.md***: Szokásos „olvass el” markdown fájl, ami minden GitHub repository-ban megtalálható.
- ***requirements.txt***: Az alkalmazás futtatásához szükséges pip csomagok listája. Ezt a fájlt megadva egyetlen parancs segítségével telepíthetjük az összes csomagot.

3.2.2. Beolvasás és kimenet

A képek beolvasására több lehetőségünk van. Az első lehetőség webkamerával valós időben rögzíteni képeket, majd az algoritmus eredményét folyamatosan megjeleníteni a képernyőn. Ezt a funkciót a cam_app.py valósítja meg.

A másik lehetőség háttértárról olvasni be előre elkészített képeket. Ebben az esetben az eredményeket opcionálisan megjeleníthetjük egymás után egyesével a képernyőn vagy egyszerre lementhetünk minden eredményt a háttértárra. Ezt a funkciót az img_app.py valósítja meg.

Az alkalmazás a képeket mindkét esetben a NumPy könyvár *ndarray* osztályának példányaként tárolja.

3.2.3. Kártyák detektálása

Az alkalmazás legfontosabb függvénye a *detect_playing_cards* függvény, ami a playing_card_detection.py fájlban található. Ez a függvény valósítja meg a tervezési fázisban felvázolt algoritmus szegmentáció, transzformáció, osztályozás és kimenet lépéseit. A függvény bemenete és kimenete egy 3 színcsatornával rendelkező RGB kép. Az algoritmus minden detektálással kapcsolatos információt rögzít a kimeneti képen. Ide tartozik a detektált kártyák kontúrjának kiemelése, és a kártyák megnevezésének kiírása az osztályozás eredményének biztosságával együtt. Az alábbi 1. ábrán egy teszt kép látható a függvény bementére és a bemenetre adott kimenetére.

1. ábra: Példa a kártya detektáló algoritmus bemenetére és kimenetére

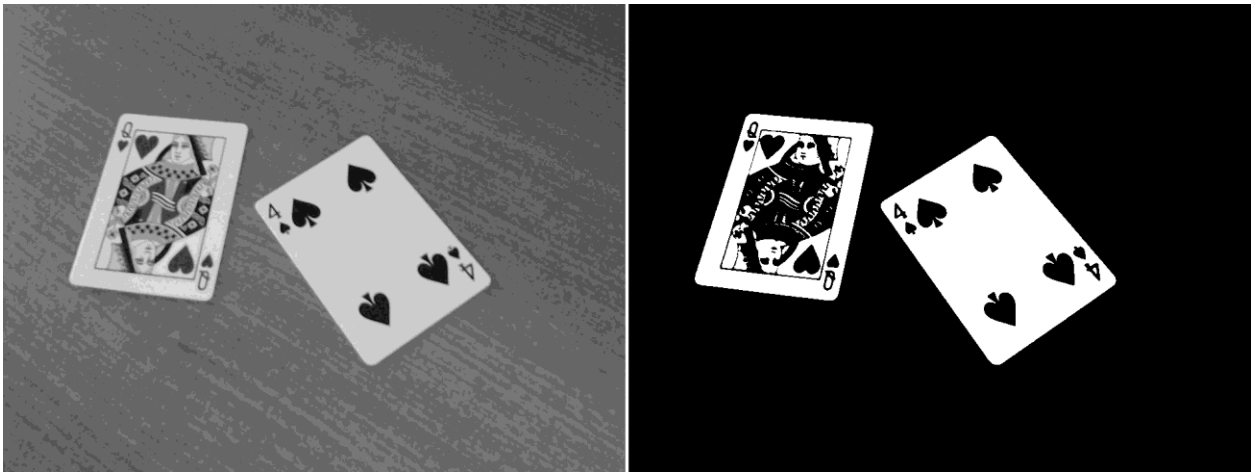
A következő fejezetek részletesen ismertetik az algoritmus egyes lépéseinek implementációját. Az algoritmus vegyesen tartalmaz különböző könyvtárakban található, előre megírt függvényeket és osztályokat, illetve saját megoldásokat. A külső könyvárakból felhasznált

legfontosabb függvények működését az elméleti háttérről szóló fejezet taglalja, a saját megoldások pedig az implementációs fejezetben kerülnek bemutatásra.

3.2.4. Világos képpontok maszkolása

Beolvasást követően az algoritmus készít egy szürkeárnyalatos másolatot az eredeti bemenő képről. A kártyák alapszínének és a háttér színének megkülönböztethetőségéről szóló megkötés eredményeként elmondható, hogy a szürkeárnyalatos képen a kártyák alapszínéhez tartoznak a kép legmagasabb intenzitású képpontjai, így azok maszkolásával szegmentálni tudjuk a kártyák alapját jelképező képpontokat környezetüktől. Ezt a feladatot végzi el a *create_white_mask* függvény.

Elsőként meghatározzuk a legmagasabb intenzitású képpont értékét a NumPy könyvtár *max* függvényével. Ezután elkészítjük a maszkot az OpenCV *inRange* függvényével, melynek paraméterei a szürkeárnyalatos kép, a legmagasabb képpont intenzitás 75%-a, és a legmagasabb képpont intenzitás. Az 2. ábrán látható példa a *create_white_mask* függvény bemenetére és kimenetére.



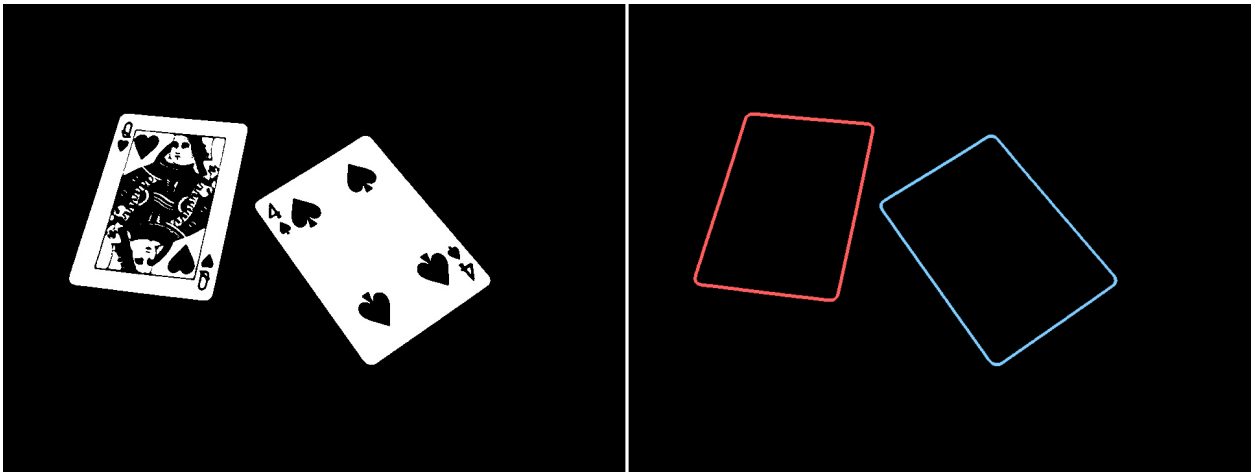
2. ábra: A *create_white_mask* függvény működésének szemléltetése

3.2.5. Kártyák körvonalának kinyerése

Az előző lépés után rendelkezünk a kártyák alapjainak maszkjával. A francia kártyák tulajdonságaiból adódóan észrevehető, hogy minden kártya szélső pereme azonos színű a kártya alapszínével, semmilyen grafikai elem nem szakítja meg a kártya szélső peremén található alapszínt. Ebből arra a következtetésre juthatunk, hogy a maszk körvonala minden esetben megegyezik a képen lévő kártyák körvonalával. Ezt kihasználva nyerhetjük ki a maszk segítségével a kártyák külső körvonalát, amit az OpenCV *findContours* függvénye végez. A függvény paraméterei a maszk, a mód paraméter, amit külső körvonalak kinyerésére állítunk, és a módszer paraméter, amit egyszerű lánc közelítésre állítunk. A függvény eredménye egy lista, ami

tartalmazza az összes képpontot, amik a kártyák körvonalaihoz tartoznak. Több kártya esetén a képpontokat nem egy folytatólagos listában tároljuk, hanem minden kártyához külön képpont lista tartozik, melyeket egy felsőbb szintű közös lista fog össze.

A 3. ábrán látható a függvény maszk bemenete és a kiszámított körvonalak képpontjai vizuálisan szemléltetve.



3. ábra: A *findContours* függvény működésének szemléltetése

A képeken különböző zajok hatására megjelenhetnek kisebb körvonalak is, amelyek jellemzően egyik kártyához sem tartoznak. Ezeket minden egybefüggő körvonalon végig iterálva tudjuk kiszűrni a körvonal területe alapján. Egy körvonal területét az OpenCV *contourArea* függvényével számíthatjuk ki. A függvény egyetlen paramétere az éppen vizsgált körvonal képpontjainak listája, kimenete pedig a körvonal képpontokban megadott területértéke. Amennyiben egy körvonal területe nem haladja meg az eredeti kép területének 2,5%-át, úgy a körvonalat hamis pozitívnak nyilvánítjuk, és a továbbiakban nem kezeljük egy kártya körvonalaként.

3.2.6. Kártyák sarokpontjainak kinyerése

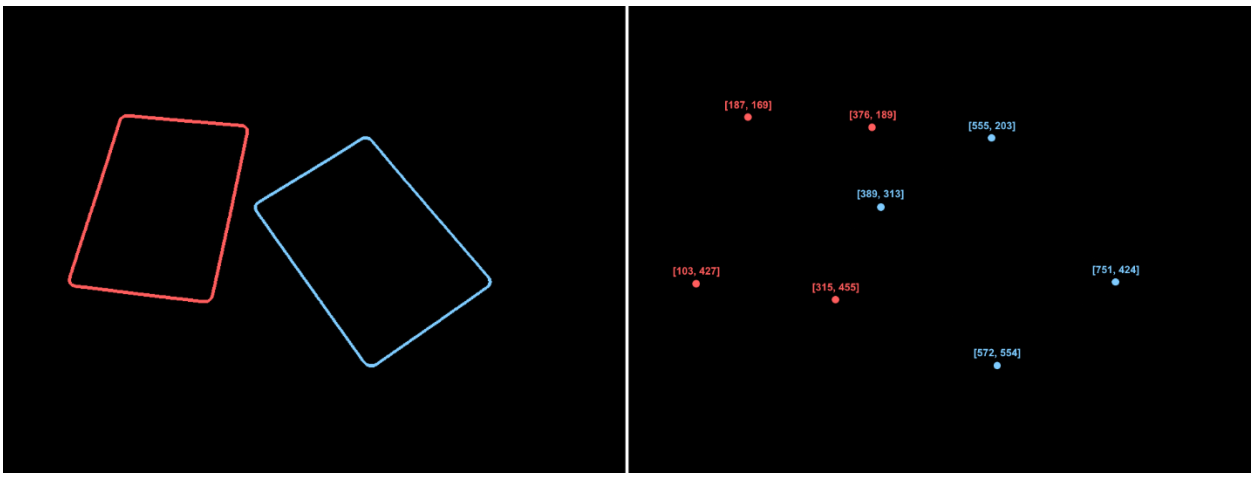
Az elkövetkező transzformációs lépések előkészületeként meg kell határoznunk minden kártya sarokpontjainak helyzetét és azok sorrendjét. Ehhez segítséget nyújthat az előző lépésben kiszámított kártya körvonalak listája, ugyanis egy körvonal képpontlistájában szerepelnie kell a keresett sarokpontoknak is. Azonban a sarokpontok pontos helyzete nem egyértelmű, hiszen a kártyák sarkai le vannak kerekítve. Ennek a problémának kiküszöbölésére használható az OpenCV két függvényének együttese, az *arcLength* függvény, valamint *approxPolyDP* függvény.

Az *arcLength* függvény kiszámítja egy kártya kerületének értéket, amennyiben paraméterként megadjuk a kártya körvonalát, és a körvonal zártságára vonatkozó paramétert igazra állítjuk. Ezután az *approxPolyDP* függvény segítségével megbecsülhetjük a körvonalra leginkább illeszkedő sokszög sarokpontjait. Az *approxPolyDP* paraméterei a körvonal, a közelítés

pontosságára vonatkozó epszilon paraméter, mely értéke a kiszámított kerület 2%-a, továbbá az előző függvényhez hasonlóan itt is megadjuk, hogy zárt körvonalról van szó. A függvény kimenete egy lista, ami a becsült sarokpontokat tartalmazza. Kártyák esetén a megközelített sokszögnek négyszögnek kell lennie, ezért csak azokat az eredményeket fogadhatjuk el, ahol a listában pontosan 4 sarokpont szerepel. Minden egyéb esetben nem tekintjük a vizsgált sokszöget és körvonalat kártyának.

A transzformációs lépések miatt ügyelnünk kell arra, hogy a kártya sarokpontjainak sorrendje állandó legyen a listában. Az *approxPolyDP* függvény tesztelését követően kiderült, hogy ez a feltétel nem teljesül. A problémát a sarokpontoknál találkozó két oldal hosszának vizsgálatával küszöbölhetjük. Ha a listában elsőként és másodikként szereplő képpontok távolsága kisebb, mint a listában elsőként és utolsóként szereplő képpontok távolsága, akkor képpontok sorrendjét 1 indexel visszafelé kell csúsztatnunk. Két pont euklideszi távolságát a NumPy lineáris algebrához kapcsolódó gyűjteményén belül a *norm* függvénnyel számíthatjuk ki hatékonyan. Az indexek elcsúsztatását szintén a NumPy könyvtárból származó *roll* függvénnyel tehetjük meg legegyszerűbben.

A sarokpontok kinyeréséhez szükséges összes lépést a *find_contour_corners* függvény foglalja magába. A függvény bemeneti paramétere egy kártya körvonalához tartozó képpontok listája, kimenete a megfelelő sorrendbe rendezett 4 sarokpont listája. Bemenetének és kimenetének szemléltetése a 4. ábrán látható.



4. ábra: A *find_contour_corners* függvény működésének szemléltetése

3.2.7. Kártyák madártávlati nézetre igazítása

Az algoritmus eddigi lépéseihez hasonlóan a soron következő jelen lépés célja szintén a képeken található információk kinyerése, azonban ezúttal ezt a célt az eredeti kép jelentős megváltoztatásával érhetjük csak el. Ahhoz, hogy a képeken lévő kártyákat a későbbiekben osztályozni tudjuk, ki kell nyernünk a kártyákhoz tartozó képpontok azon részeit, amelyek alapján a kártya egyértelműen beazonosítható. Francia kártyák esetén a keresett tulajdonságok a kártya bal

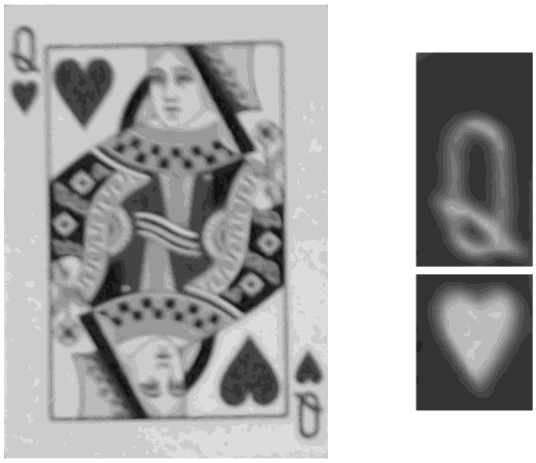
felső és jobb alsó sarkában találhatók. A feladat nehézségét az jelenti, hogy felvételt készítő eszköz és a kártyák egymáshoz viszonyított helyzete torzíthatja a keresett tulajdonságok megjelenését, ezzel közel lehetetlenné téve a pontos azonosítást. Azonban a kártyák eredeti tulajdonságainak ismeretével a vizsgálat során lehetőségünk nyílik a kártyák megjelenésének mesterséges helyreigazítására. Ezt a feladatot látja el a *create_birds_eye_view* függvény, amely segítségével külön-külön igazíthatjuk egységes madártávlati nézetre a képeken szereplő kártya példányokat.

A függvény bemeneti paraméterei a szürkeárnyalatos kép inverze, a kártya 4 sarokpontjának listája és a transzformáció kimenetének felbontása. A kimeneti felbontás meghatározásához felhasználjuk azt az ismeretet, hogy a francia kártyák alakja egy lekerekített sarkú téglalap, melynek oldalhosszai 2,5” és 3,5”. A méretarányokat megtartva tehát az algoritmusban 500x700 pixel képfelbontást használunk a függvény paramétereként. A függvényen belüli transzformációs lépéseket az OpenCV könyvtár *getPerspectiveTransform* és *warpPerspectie* függvényei végzik. A *create_birds_eye_view* kimenete egy madártávlati nézetre igazított kép, melynek teljes egészét egy kártya tölti ki. A függvény hívását megelőző színinverzió célja, hogy a kártya alapszíne sötét, a kártyán található grafikai elemek világos képpontokként jelenjenek meg. Mindez azt hivatott jelképezni, hogy a kártyán lévő szimbólumokra koncentrálunk nem a szimbólumok hátttereként szolgáló alapszínre. Az 5. ábrán a függvény bemenetét és kimenetét szemléltető példák láthatók.

5. ábra: A *create_birds_eye_view* függvény működésének szemléltetése

Miután minden detektált kártya megjelenése egységes, kinyerhetjük az osztályozáshoz szükséges információkat. Ahogy a fejezet első felében is elhangzott, a kártyák keresett tulajdonságai a kártya bal felső sarkában találhatóak. Az első ilyen tulajdonság a kártya rang értéke, amit egyszerűen kivághatunk a transzformált képből, mivel a rangot jelképező szimbólum mindig a kép azonos területén belül helyezkedik el. Szintén így járhatunk el a második keresett tulajdonság kinyerésekor, amivel a kártya színét határozhatjuk meg. A kivágott képrészletek méretét tapasztalati úton választottam 60x110 és 60x70 pixel méretűre. Ezt a módszert alkalmazva előfordulhat, hogy a szimbólumok kis mértékben el vannak forgatva, vagy egy részük kilóg a képkivágásból. Ennek oka az előző lépésben történő sarokpontok tökéletlen kiszámítása.

Szerencsére még így is nagyságrendekkel alacsonyabb eltérés figyelhető meg egy-egy kivágott szimbólum megjelenésében, mint ha az eredeti képről vágnánk ki a kártya ugyanezen területét. A szimbólumok kivágása a 6. ábrán látható.



6. ábra: A transzformált képből kivágott szimbólumok inverze

3.2.8. Kártyákon lévő szimbólumok osztályozása

A francia kártyákat 13 féle rang és 4 féle szín szimbólum segítségével azonosíthatjuk. Ezek a szimbólumok közel azonos mérettel és jól megkülönböztethető formával rendelkeznek csakúgy, mint a klasszikus MNIST adatkészletben található kézzel írt számjegyek. E párhuzam által inspirálva a szimbólumok osztályozási problémájának megoldásához konvolúciós neurális hálózatok használatát választottam, amit tipikusan alkalmaznak az imént említett az MNIST adatkészlet osztályozásához is. A kártya szimbólumok két típusát két különböző hálózat osztályozza. Ennek elsődleges oka a szimbólumokat tartalmazó képkivágások eltérő mérete, miszerint a rangokat tartalmazó képek 60x110, a színeket tartalmazó képek 60x70 pixel felbontásúak. A rangokat osztályozó hálózat modelljének forráskódja a 7. ábrán látható. A modellnek összesen 584333 tanítható paramétere van. A színek modellje a rangok modelljéhez képest a bemenő réteg méretében, valamint a kimenő réteg méretében tér el, igazodva a képkivágások méretéhez és az osztályok számához.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(110, 60, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(13, activation='softmax')
])
```

7. ábra: A kártyák rang szimbólumait osztályozó konvolúciós neurális hálózat felépítése

A modellek 10 rétegből állnak. Az első 6 réteg felváltva tartalmaz 3 konvolúciós réteget és 3 összevonó réteget. Ezeket a rétegeket egy köztes kisimító réteg köti össze a végső 3 teljesen összekapcsolt neuronréteggel. Minden réteg paraméterei megfigyelhetőek az 7. ábrán, működésük leírása az elméleti hátteret bemutató fejezetben olvasható. A hálózat tanításához rengeteg tanító mintára van szükség. Ezeket a mintákat a record.py script-tel rögzítettem, majd a process.ipynb notebook-kal dolgoztam fel. Szimbólumonként 3200 mintakép készült, vagyis összesen 54400 darab, amiből 41600 a rangok osztályit, 12800-on színek osztályit tartalmazzák. Mindkét hálózat esetén az adatkészletet 9:1 arányban osztottam el tanító és teszt adatokra. A tanítás során a batch méretet a TensorFlow alapértelmezett értékén hagytam, ami 32. Az epoch-ok számát 10-re állítottam. Hibafüggvényként az osztályozási feladatokhoz tipikusan használt kategorikus keresztentrópia függvényt választottam. Végezetül az Adam tanító algoritmust használtam a hálózat tanítására.

A megalkotott konvolúciós neurális hálózat bemenete egy képkivágás, amely egy kártyán található szimbólumot tartalmaz. Kimenete az utolsó rétegben található softmax függvény által kiszámított valószínűségi eloszlás. Az algoritmus az eloszlás értékeit 2 tizedesjegy pontossággal kerekíti, és a legnagyobb értékéhez tartozó osztályt tekinti a szimbólum valódi osztályának.

A kész modelleket a TensorFlow *save* függvényével menthetjük a háttértárra. Ebből adódóan az alkalmazás úgy épül fel, hogy az algoritmus a lementett modelleket olvassa be, majd használja a kártyák azonosításához. Az ilyen módú felépítésből következik, hogy a modellek paraméterei az algoritmus kódjának módosítása nélkül, teljesen függetlenül változtathatóak. Kivétel ez alól a bemeneti és kimeneti réteg, amelyek szoros kapcsolatban állnak az algoritmus működésével. Szintén előnyös, hogy a lementett modelleket nem kell minden alkalommal újra tanítani, amikor lefuttatjuk az algoritmust.

3.2.9. Eredmények megjelenítése

Az algoritmus utolsó lépése az eredmények megjelenítése. Itt a cél feltűntetni minden hasznos információt az eredeti képen, annak jelentős megváltoztatása nélkül. A gyakorlatban ezt az

információk rétegszerű megjelenítésével érhetjük el ügyelve arra, hogy az eredeti képen minden fontos részlet felismerhető maradjon.

Az előző lépések eredményeiből ismerjük a képen lévő kártyák körvonalát, rangját és színét. A kártya példányok sikeres felismerését a körvonalak kiemelésével jelképezhetjük. Körvonalat az OpenCV *drawContours* függvényével helyezhetünk a képre. Emellett javít a megjelenítésen, ha a kártyához tartozó képpontok színárnyalatát kis mértékben megváltoztatjuk. Ezt úgy tehetjük meg egyszerűen, hogy a kártya körvonalával megegyező alakú síkidomot hozunk létre a körvonal kitöltésével és ezt az alakzatot helyezzük súlyozva az eredeti képre. Az alakzat létrehozása az OpenCV *fillPoly* függvényével oldható meg, majd ezt az *addWeighted* függvénnyel tehetjük a képre. A kártyák rang és szín tulajdonságait szöveges formában jeleníthetjük meg a kimeneti képen. Ahhoz, hogy átlátható legyen melyik szöveg melyik kártyához tartozik, a szöveg a kártya bal felső sarkától kezdődik és olvashatósági szempontból mindig vízszintesen helyezkedik el. A szöveg tartalmazza a kártya rangjának és színének osztályát, valamint az osztályokhoz tartozó biztossági értéket, amit a neurális hálózat határozott meg. Ezt a szöveget az OpenCV *putText* függvényével jeleníthetjük meg a képen. A körvonal és a szöveg színét érdemes úgy választani, hogy minden kártya kiemelő színe eltérő legyen a képen, de tulajdonságaikban hasonló kártyáknak hasonló legyen a kiemelő színe is. Feltételezve, hogy a képek 8 bites színmélységűek, egy-egy színcsatornát felosztottam annyi részre, ahány különböző osztálya lehetséges az adott szimbólumtípusnak. A megjelenített kiemelő színt a kártya szimbólumainak osztálya szerint külön keveri ki az alkalmazás minden kártya példány esetén.

A felsorolt lépéseket a *create_card_label* függvény végzi, melynek bemeneti paramétereként az eredeti képet, a vizsgált kártya körvonalát, sarokpontjait, rangját, a rang biztosságát, színét és végül a szín biztosságát kell megadni. A függvény kimenete az eredeti kép módosított változata, amelyen a többi bemeneti paraméternek megfelelő kártya példány kiemelve és címkézve jelenik meg. Minden lépést összesítve a 1. ábrán látható az algoritmus bemeneteként megadott kép és a kimeneteként előállított kép.

3.3. Tesztelés

3.3.1. Neurális hálózatok tesztelése

A neurális hálózatok teszteléséhez az adatkészletet két részre osztottam. Az első rész az adatkészlet 90%-át tartalmazza, ami a hálózat tanítására szolgál. A második rész maradék 10%-át tartalmazza, ami a hálózat tesztelésére szolgál. A felosztást megelőzően az adatkészletet véletlenszerűen összekevertem, hogy minden osztályhoz több minta is tartozzon a tesztkészletben. Miután megtörtént a hálózat tanítása, megkezdődhet a hálózat teljesítményének tesztelése. A tesztelési folyamat lehetővé teszi számunkra, hogy lássuk, milyen teljesítményre képes a hálózat új, korábban nem látott minták vizsgálata esetén. Sok különböző mérőszám létezik, ami neurális hálózat teljesítményének mérésére használható. Néhány gyakori mérőszám a pontosság és a hibaérték. A hibaérték annak mértéke, hogy a hálózat mennyire képes helyes előrejelzéseket biztosítani egy adott adatkészletre vonatkozóan. Értékét az előrejelzés és az elvárt kimenet közötti

különbség meghatározásával számíthatjuk ki. A kisebb hibaérték jobb teljesítményt jelent. A másik mérőszám a pontosság, ami arra vonatkozik, hogy a hálózat milyen arányban határozza meg a minták helyes osztályát. Ebben az esetben a nagyobb értékek jelentenek jobb teljesítményt. A rang és szín szimbólumokat osztályozó hálózatok teszteredményei a következők:

	Rang hálózat	Szín hálózat
Hibaérték	0,01065	0,00028
Pontosság	99,76 %	100 %

Látható, hogy a hálózati struktúrák komplexitásából és az adatkészlet méretéből adódóan mindkét modell elfogadható teljesítménnyel képes elvégezni az osztályozási feladatokat. Ennek ellenére számítanunk kell arra, hogy a hálózatok teljesítménye erősen függ a környezeti tényezőktől, így nem minden szituációban lesznek képesek ilyen mértékű teljesítményt nyújtani.

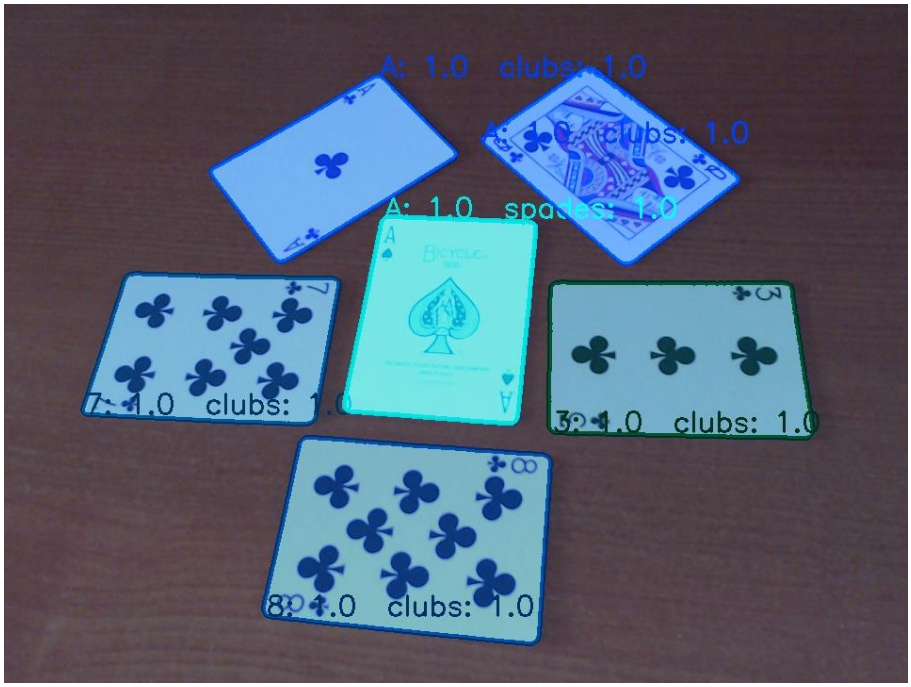
3.3.2. Teljes algoritmus tesztelése

A neurális hálózat modellek tesztelésén kívül fontos a teljes algoritmus tesztelése is. Ilyenkor a hálózatok mellett minden egyéb képfeldolgozási lépés befolyásolja a teljesítményt. A fejlesztés folyamata közben ezeket a képfeldolgozási lépéseket külön-külön teszteltem a webkamerát igénylő script (img_app.py) futtatásával. Ennek a módszernek az előnye, hogy valós időben azonnal látok visszajelzést arról, hogy hogyan reagál az adott lépést megvalósító kódrészlet a környezeti változásokra. A tesztelési módszer gyorsasága és testreszabhatósága lehetővé teszi, hogy tapasztalati úton hamar átfogó képet kapjak az algoritmus aktuális teljesítményéről, és eszerint változtassam meg a kód paramétereit és működését. A módszer hátránya, hogy a teljesítmény tapasztalati úton határozható meg mérőszámok bevezetése és statisztikai adatok gyűjtése nélkül, valamint egy-egy teszt kísérlet nem megismételhető. Éppen emiatt a valós idejű webkamera módszer nem alkalmazható az elkészült végleges algoritmus teljesítményének felmérésére, ahol mért teszteredmények dokumentálása a cél.

A teljesítmény dokumentálásához manuálisan készítettem és kiértékeltem 50 darab képet, amiken legalább egy francia kártya látható több különböző szituációban. A képek közül 11 elérhető a repository test mappájában eredeti és feldolgozott képek formájában egyaránt. A teszt képek készítésekor figyelembe vettem, hogy a paklinak minden típusú kártyája legalább egy alkalommal szerepeljen. Emellett a képek változatosságára is ügyeltem, hogy fellelhetőek legyenek az algoritmus erősségei és gyengeségei. Az 50 képen összesen 162 kártya példány volt jelen. A 162 kártya példány közül 120 szegmentálása és osztályozása történt helyesen, ami 74% körüli pontosságot jelent a tesztadatkészletre. A hibás esetek közül 7 alkalommal egyáltalán nem sikerült detektálni kártya jelenlétét, melyből minden eset a 3.1.3. fejezet megkötéseihez kapcsolódik. A maradék 35 kártyának a pontos osztályozása volt sikertelen, ahol 14 alkalommal a rang, 8 alkalommal a szín meghatározása volt a probléma. 13 alkalommal azonban egyik tulajdonság osztályozása sem volt sikeres.

A tesztelés összegzése a következő:

- A kártya példányok szegmentációja kizárólag a megkötések figyelmen kívül hagyása mellett sikertelen
- A sikeresen szegmentált kártyák típusának azonosítása ideális körülmények között elfogadható pontossággal működik, de itt is ügyelni kell a megkötésekre
- A sikertelen osztályozás leggyakoribb okai az azonosításhoz használt szimbólumokat érintő zajokból erednek, jellemzően ilyenkor emberek számára is rendkívül nehezen felismerhetőek a kivágott szimbólumok
- Ha minden körülmény ideális, akkor is ritkán előfordulhat, hogy a neurális hálózat magas biztossággal és hibásan osztályoz szimbólumokat



8. ábra: Az algoritmus átlagos teljesítményére jellemző tesztkép

4. Felhasználói leírás

A repository nem tartalmazza a sokezer képből álló tanító adatkészletet és az 50 MB-ot meghaladó modell fájlokat. Ebből adódóan ezeket az alkalmazás használata előtt manuálisan kell előállítanunk, melyekhez azonban minden script megtalálható a repository-ban. Ahhoz, hogy futtatni tudjuk a scripteket, telepítenünk kell a Python-t, a Jupyter Notebook-ot és az összes pip csomagot, ami a requirements.txt-ben található.

4.1. Előkészítő lépések

4.1.1. Tanító képek előállítása

A tanító képeket webkamerával rögzíthetjük a record.py futtatásával. A script 5 paramétert vár:

- Eszköz ID (általában 0)
- Kép szélessége
- Kép magassága
- Képek száma
- Kimeneti mappa

A folyamat során érdemes ugyanazt a webkamerát használni, mint amivel az alkalmazás kártyadetektáló algoritmusát is használni fogjuk. Emellett érdemes úgy strukturálnunk a tanító képek rögzítését, hogy egy alkalommal csak egy szimbólum rögzítésére koncentrálunk. Például kezdésként csak olyan képeket rögzítsünk, amelyek egyetlen 2-es rangú kártyát tartalmaznak. A rögzítés kimenetéhez ebben az esetben a „2” mappanevet adjuk meg. A leírt folyamatot ismételjük meg minden szimbólummal, ami 13 rangokkal kapcsolatos képsorozatot és 4 színekkel kapcsolatos képsorozatot jelent.

4.1.2. Tanító képek feldolgozása

A rögzített képeket a process.ipynb celláinak futtatásával tehetjük meg. Ha az előző lépésben máshogy neveztük el a mappákat a leírtakhoz képest, itt meg kell változtatnunk a második cella tartalmát. A futtatáshoz a notebook fájlt tartalmazó mappába be kell másolnunk az src/image_manipulation.py fájlt, ami a képfeldolgozó függvényeket tartalmazza. A képek feldolgozása után 4 npy kiterjesztésű fájlt kell kapnunk, melyek közül 2 a feldolgozott képeket tartalmazza, a másik 2 pedig a képekhez tartozó címkéket.

4.1.3. Modellek tanítása

A neurális hálózat modelleket az előző lépésből származó fájlok segítéségével taníthatjuk. A tanítást a models mappában található 2 notebook fájllat tehetjük meg, melyek mellé be kell

másolnunk az npy kiterjesztésű fájlokat. A tanítást követően nincs más teendőnk, az alkalmazás készen áll a használatra.

4.2. Algoritmus futtatása

4.2.1. Webkamera mód

Ahhoz, hogy a kártyadetektáló algoritmust webkamerával futtassuk, a cam_app.py-t kell elindítanunk. Indításhoz a következő paraméretek megadása szükséges:

- Eszköz ID (általában 0)
- Kép szélessége
- Kép magassága

A kép méretek megadásánál ügyelnünk kell arra, hogy a webkameránk támogassa a megadott felbontást. Amennyiben támogatja, a képernyőn ebben a felbontásban jelenik meg egy ablak az algoritmus kimenetével. A programot bármelyik billentyű megnyomásával bezárhatjuk.

4.2.2. Fájlbeolvasás mód

Ha webkamera használata helyett előre elkészített fájlokat szeretnénk beolvasni, arra is van lehetőség az img_app.py futtatásával. Indításhoz a következő paraméretek megadása szükséges:

- Bemenő képek mappája
- Kimenő képek mappája (opcionális)

A bemenő képek mappájából az algoritmus beolvassa a képeket és feldolgozza őket. A kimenetet két módon kaphatjuk meg. Amennyiben nem adunk meg kimeneti mappát, úgy minden egyes kép megjelenik egy ablakban a képernyőn. Ilyenkor a programot az ESC billentyűvel zárhatjuk be. A soron következő képre továbblépni a ESC billentyűn kívül bármelyik billentyű megnyomásával lehetséges. Ha megadjuk a kimeneti mappa elérési útvonalát, úgy a feldolgozott képek abba a mappába kerülnek mentésre.

Irodalomjegyzék

[1] OpenCV hivatalos dokumentáció - https://docs.opencv.org/4.x/index.html
[2] TensorFlow hivatalos dokumentáció - https://www.tensorflow.org/api_docs/python/tf/all_symbols