Feladatkiírás

A szakdolgozat elkészítése során a szakdolgozó megismeri a mesterséges intelligencia által nyújtott lehetőségek gyakorlati alapjait. Ezen ismeretséget felhasználva pedig megvizsgálja a terület alkalmazhatóságát egy olyan képi objektumfelismerési feladatban, ahol az entitásként megjelenő objektumok egy képen azonos osztályba tartoznak, de különböző képeken különböző osztályban lévő entitások fordulhatnak elő. A szakdolgozat végén meg kell vizsgálni, hogy mennyire alkalmazható korunk egyik legújabb eszköze, a mesterséges intelligencia egy globális statisztika felállításához a hagyományos módszerekkel szemben.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Absztrakt objektumok detektálása gépi tanulási modellel előre ismert entitás mintázat alapján

- ***A megadott feladat megfogalmazása:***

A feladat egy prototípus elkészítése képen elhelyezkedő objektumok felismerésére és megszámlálására. Az objektumok különbözőek lehetnek, de egy képen egyszerre csak egy típusú entitás jelenhet meg. A feladathoz gépi tanulási módszereket kell alkalmazni annak érdekében, hogy a végeredmény kellően absztrakt legyen és, hogy a modell architektúrája újra felhasználhatóvá váljon.

- ***A megoldási mód:***

Megoldásomban egy Autoencoder MI architektúrát alkalmazok, amely képes előállítani a bemenő képet azáltal, hogy megtanulja annak egyedi jellemzőit. Ezen jellemzők azért fontosak, mert megtanulásukkal szegmentálásra képes modell is elkészíthető az architektúrából, de kellőképpen absztrakt az implementáció, hiszen a bemenő képeken lévő objektumok típusa és mintázata nincs megkötve.

- ***Alkalmazott eszközök, módszerek:***

Eszközként a „Google Brain” csapat által biztosított ML könyvtárat, nevezetesen a Tensorflow-t használtam, mely egy felhő környezetben is elérhető, az ingyenes Google Colab nevű Jupyter notebook környezetben. Emellett egy lokális python környezetet is elkészítettem, mely segítségével könnyebben tudtam tesztelni az egyes megoldások futási idejét és hatékonyságát.

- ***Elért eredmények:***

Az implementált architektúra képes olyan modellek tanítására és elmentésére, amelyek segítségével a bemenő képen lévő azonos objektumok felismerhetők egy egyszerűsített környezetben. A modellek egy látens dimenzióban rejtik el a megtanult jellemzőteret, amely egyszerűségéből adódóan remekül használható arra, hogy egy utófeldolgozást követően megszámlálhatóvá válljanak a felismert objektumok.

- ***Kulcsszavak:***

Gazdasági előrejelzés, globális gazdasági statisztikák, mesterséges intelligencia, absztrakt objektum detektálás

Tartalomjegyzék

BEVEZETÉS

Napjainkban a technológia fejlődése úgy tűnhet, hogy megállíthatatlan tempóban halad és ehhez nagyban hozzájárul a globalizmus és az információ könnyű és gyors terjedése. Mindazonáltal ez a folyamat magába foglalja azt is, hogy megjelennek olyan technológiai csodák, mint a mesterséges intelligencia, amely bár igen jövőbemutató, a szakembereknek be kell vallania, még mindig gyerekcipőben jár. Ez a terület azonban számomra is igen vonzó aspektusokat tartalmaz, amely elegendő volt ahhoz, hogy megszülethessen ez a dolgozat.
Jelen dokumentumban egy olyan témát kell körüljárnunk, amely gépi tanulási módszerek alapjainak megismerése mellett egy prototípust próbál előállítani képi objektumok felismerése céljából. Az objektumok absztrakt jellemzőit egy MI-modellel kell feltérképeznünk, ezáltal a tanulás mintafüggetlenné tehető. Az objektumok mintázata kötött, tehát a tanulás során bemenő paraméterként megadható, de a mintázatok modellenként változhatnak. A gépi modellnek fel kell ismernie a képen lévő objektumokat és meg kell őket számlálnia, vagy megszámlálható kimeneti adathalmazt kell előállítania.
A prototípus segíthet egy olyan koncepció implementálásában, amely segítségével a Földünkön lévő fontos gazdasági objektumok felismerhetők és mennyiségük megbecsülhető. Ez az adat segíthet határok nélküli statisztikát készíteni, amellyel bizonyos aspektusból folyamatosan követhetővé válik a nagy gazdasági folyamatok következménye.

1. KÉPALKOTÓ LEHETŐSÉGEK

Ahhoz, hogy egy modellt előállítsunk szükség van egy olyan adathalmazra, amely megfelelő információtartalommal rendelkezik a megtanulni kívánt viselkedés implementálásához. Ez jelen feladatban azt jelenti, hogy szükségem volt egy olyan képekkel teli adatbázisra, amelyen egy osztályba tartozó objektumok találhatók. Az objektumok felismerése ezután több módon történhet, de erre a későbbiek során térnék ki. Ebben a fejezetben azt kívánom megörökíteni, hogy milyen módszerekkel próbálkoztam egy felhasználható adatbázis megszerzését illetően.

1.1. Az első ötletek

Többször hallottam már életem során, hogy egy szoftver mit sem ér a mögötte lévő adatbázis nélkül. Jelenlegi csekély tapasztalataim azt sugallják, hogy ennek a megfigyelésnek csak

akkor van igazságtartalma, ha egy szoftver olyan adatokkal dolgozik, melyek integritása nem sérülhet, legtöbbször tehát adatorientált szoftverek körében lehet igaz az előbbi állítás.
Ez a gondolat már többször is felmerült bennem és ezen feladat keretein belül igyekeztem egy olyan megoldást keresni, amellyel bizonyítható az igazam. Tudtam, hogy ha egy végtelen nagyságú értelmezési tartományon kell dolgoznom, akkor nem fogok tudni összeszedni annyi teljes értékű adatot, amellyel reprezentálni tudom a különböző természeti előfordulásait egy objektumnak. Ezért egy olyan megoldásban gondolkodtam, ahol nincs szükség pontos adatokra, inkább csak az entitások jellege a fontos. Erre a gondolatmenetre alapozva akár valamilyen automatizált módszerrel elő is tudnám állítani a tanításhoz szükséges adatokat, figyelve arra, hogy azok tartalmazzák a szükséges természeti körülményeket, amelyek fontos tanulási pontok lehetnek a végeredmény szempontjából.
A legelső ötletem az volt, hogy kiöntök egy csomag borsót vagy babszemet vagy bármilyen más kicsi tárgyat egy viszonylag tiszta felületre, hogy elő tudjak állítani egy adatbázist azonos objektumokat tartalmazó képekből. Ez az ötlet elsőre olyan jónak tűnt, hogy majdnem elkészítettem egy erre a célra szánt mobilalkalmazást C# nyelven Xamarin segítségével. A tesztelés folyamán azonban rájöttem, hogy ez a megoldás azért nem lesz jó, mert ha nagyon sok képet kell készítenem, például többszáz- vagy akár többezer adatra lesz szükségem, akkor ez a folyamat hosszú ideig is elnyúlhat és egy prototípus enélkül is elkészíthető. Így tehát erről a módszerről lemondtam az időtényező miatt, de nem szabad figyelmen kívül hagyni azt a tényt, hogy olyan jellemzőhalmazt is képesek vagyunk összegyűjteni ilyen képekről, amelyek reprezentálhatják a rajta lévő objektumok rendezetlenségét, fényviszonyait, természetes zajait és egyedül a kamera képességei szabhatnak korlátot.
A második megközelítés már egy sokkal prototípus barátabb ötlet terméke, amely során csak a cél objektumokat reprezentáló alegységekből generált bemeneti képeket kívántam használni a tanítás során. Röviden arról van szó, hogy összegyűjtöttem 8-10 db napraforgófejről készült képet, melyeket véletlenszerűen elhelyeztem egy fehér háttérrel rendelkező képen kombinatorikai algoritmussal. A végső kép 8-12 darab augmentált objektumot tartalmazhat egy egyszerű háttérképen. Ütközéseket nem engedtem meg az entitások között, ugyanis ezen tény triviális mivolta nem volt elég meggyőző számomra, hogy a prototípus implementálása során során erre erőforrást szánjak, de többször is figyelembe vettem ezt, és még pár másik fontos aspektust is a tervezés során, amelyek előfordulhatnak a természetben.

1.2. Skálázási problémák

Akár generált adatokkal dolgozunk, akár természetből készült képekkel, a skálázásra gondolni kell, hiszen a bevezetőben emlegetett céltartomány hatalmas. Egy egész bolygó erőforrásait feltérképező modellt bemenő adatokkal csak úgy tudunk ellátni, ha az égitest felületét aprólékosan és graduálisan vizsgáljuk. Ez azt jelenti, hogy a végső modellt fel kell készíteni olyan szituációra, ahol a bemenő adatok között kapcsolatokat tud detektálni és végül eddigi ismeretei alapján össze tudja illeszteni a bemenő részhalmazt a kimeneten megjelenő növekményes részhalmazzal.
Ezt a problémát dolgozatom során sajnos nem tudtam figyelembe venni, de igyekeztem olyan megoldást implementálni, amely részben vagy egészben kompatibilis lehet egy olyan bővítménnyel, amely már képes felismerni a bemenetek közötti összefüggést, ezzel megfejtést kínál arra a fontos követelményre, hogy a felismert objektumok duplikációmentesen jelenjenek meg a végeredmény értékkészletén.

1.3. A prototipizált környezet

A végső környezet végül egyszerű fehér háttérképpel rendelkező, célobjektumokat tartalmazó képek halmazából áll. A képeket egy egyszerű algoritmussal python algoritmussal generáltam. Ez egy prototípus architektúra elkészítéséhez elegendő volt, de a bemenő adat a tanulás során cserélhető, gyakorlati tapasztalataim szerint néhány szuperparaméter beállítása után teljesen egyező kimenet generálható összetettebb képadatok esetén is.

2. OBJEKTUMFELISMERŐ TECHNIKÁK

Mint legelső projekt feladatom, melyben mesterséges intelligenciát szerettem volna használni, utána kellett néznem mélyebben a témának. Habár az egyetemi kurzusokon az alapokat jól megtanították, a gyakorlatorientált részében még nagyon tapasztalatlannak éreztem magam.

Ezért elkezdtem képi objektumfelismesmerési módszerek után kutatni. Számtalan gépi tanulási módszerrel és algoritmussal találkoztam, amelyekkel elméletben implementálható az alapötlet, de nem voltam meggyőződve arról, hogy ezekkel az algoritmusok könnyen meg tudnám valósítani az absztrakt képfelismerést. Néhányat már jómagam is kipróbáltam pár kisebb projekt keretein belül, de alább részletezem milyen módszereket ismertem meg kutatásom során.

- Template matching
- Mean shift algoritmus
- CNN
- R-CNN és Faster R-CNN

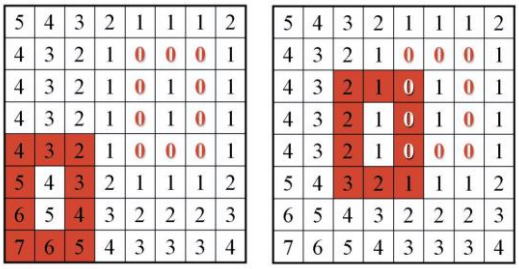
2.1. Template matching

Talán az egyik legegyszerűbb algoritmus arra a célra, hogy bizonyos objektumokat megtaláljunk egy képen. A koncepció nagyon egyszerű, fogjunk egy objektumot reprezentáló képponthalmazt, helyezzük el 0,0 pontban, majd számoljuk ki, hogy mekkora az eltérés a képpontokat leíró színértékek között. A végén toljuk el 1 képpontal az mintaobjektumunkat egyik irányba és számoljuk ki ott is a képpontok súlyait. A végeredmény az, hogy minden lehetséges pozícióban megvizsgáltuk a keresett objektum lehetőségét, és ezáltal meg tudjuk mondani, hogy mely pontokban volt a legnagyobb egyezőség, hiszen ott lesz a legkisebb az eltérés. Ezt természetesen lehet mohó módon is implementálni, ahol nem 1-el toljuk az minta objektumot, hanem egy nagyobb léptékkel, de bizonyos esetekben, ez sajnos nem tud jól működni.

2.2. Chamfer távolság

Az egyik kedvenc algoritmusom, melyet először Dr. Palágyi Kálmán kurzusain ismertem meg. Viszonylag könnyen implementálható és olyan esetben, ahol az objektumok alakzata nagyjából ugyan olyan, remekül használható. Sajnos hamar rá kellett jönnöm, hogy ha használni is szeretném ezt az eljárást, akkor is ki kell egészítenem valamilyen változóhalmazzal, amely reprezentálni próbálja az alakzat méretét, alakbeli minimális eltérését és egyéb augmentációs jellemzőit. Ez az algoritmus viszont már nem lett volna hatékony, hiszen a Chamfer illesztést nehéz változó jellemzőtérrel rendelkező entitásokra alkalmazni.

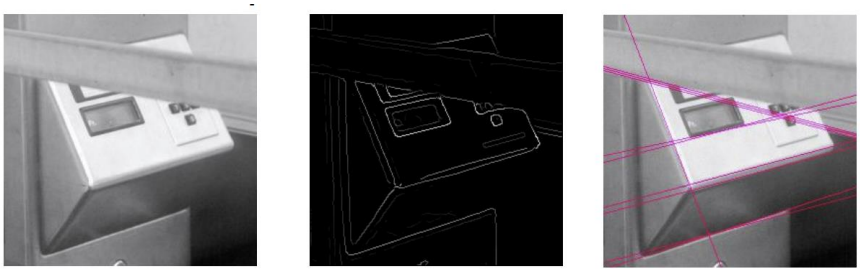
Ezt az algoritmust talán a template matching egy speciális implementációjaként is fel lehet fogni. A chamfer távolság kiszámítása egy egyszerű matematikai megközelítéssel lehetséges. A távolságot mindig a kontúrhoz képest mérjük. Azokon a képpontokon, amelyek a kontúrt tartalmazzák, a távolságot nullának számoljuk. Minden képpont a kontúrtól távolodva exponenciálisan inkrementálódik 1-el. Az algoritmust általában mohó kereséssel implementálják, a cél az, hogy a keresett objektumot helyezzük el a képen egy véletlenszerű pozícióban és haladjunk abba az irányba, amerre a keresett objektummal elfoglalt képpontokon lévő súlyok összege egyre kisebb. Ebből látható, hogy a Chamfer távolság csak akkor használható igazán, ha a képen csak egyetlen egy célobjektum van, és annak az objektumnak a kontúrja teljesen szegmentálható.



Illusztráció a Chanfer illesztésről Dr. Palágyi Kálmán prezentációjából

2.3. Hough transzformáció

Számomra ez az egyik leglélegzetelállítóbb algoritmus. Amikor részletesen megismertem, rengeteget gondolkodtam rajta, hogy hol tudnám felhasználni. Igazából nagyon sok területen alkalmas lehet, főleg akkor, ha a bemenő adatként szolgáló képen egyértelműen beazonosítható a célalakzat a kontúr alapján. A transzformáció egy olyan képi látási technika, amely segítségével az alakzatok, élek és vonalak szegmentálhatók az eredeti képről. Valószínűleg az itt említett algoritmusok használata esetén a legtöbb alkalommal egy előfeldolgozási lépésben megjelent volna a Hough transzformáció is. A probléma ezzel csak annyi, hogy maga a kontúrok ismerete jelen feladatban nem rendelkezik elég tág információhalmazzal, hiszen a fentebb már említett etalon esetben is, egy napraforgófej könnyen összekeverhető lenne egy százszorszép fejével, ha ugyanolyan méretben jelennek meg a bemeneten.



A kép illusztráció a Hough transzformációról, Kató Zoltán prezentációjából

2.4. Egyéb éldetektálási és kontúrdetektálási algoritmusok

Természetesen az éldetektálásnak számtalan módszere van, amelyek közül egyik másik algoritmust próbáltam én is mélyebben megismerni, mint Sobel és Prewitt operátor, Zero crossing detector, illetve különböző Gauss féle differenciák.

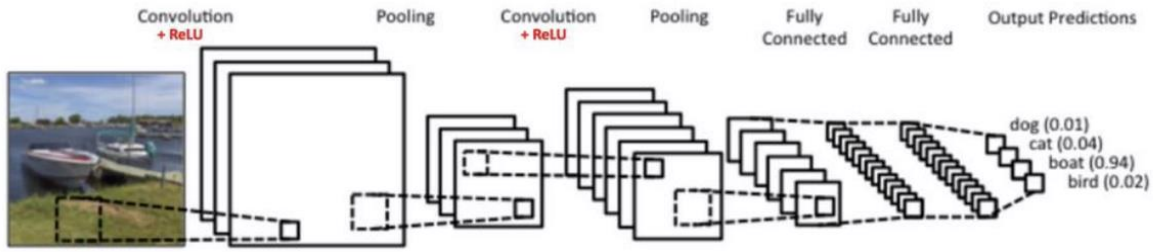
2.5. Mean shift algoritmus

Ez az algoritmus egy kifejezetten klasszifikációra tervezett módszerrel dolgozik. Iteratívan végigvezet egy kernelt a képen, amely segítségével kiszámolja az adott kernelen lévő képpontok alapján, hogy azok milyen klaszterbe sorolhatók. A futás végén látszódni fog, hogy milyen valószínűséggel tudta csoportosítani az adott kép pontjait egy adott klaszterbe.

2.6. Konvolúciós neuronhálók

A mesterséges intelligencia kontextusában talán ezzel a módszerrel találkoztam a legtöbbet. Ez a háló a mesterséges neurális hálóknak egy olyan fajtája, amely a bemenő adatot hierarchirchikusan dolgozza fel, ez természetesen azt is jelenti, hogy a háló szükségszerűen mély lesz, hiszen a képeket nem egyben dolgozza fel, hanem egyszerre csak kis részeire koncentrálva frissíti a súlyokat. Minél magasabb rétegbe érünk annál nagyobb képrészletet dolgoz fel a modell, amely azt eredményezi, hogy a képen lévő objektumokat a háló magasabb szinten képes azonosítani, annak ellenére, hogy a felbontásuk magas szinten már nem számít.

2.7. Régió alapú konvolúciós neuronhálók



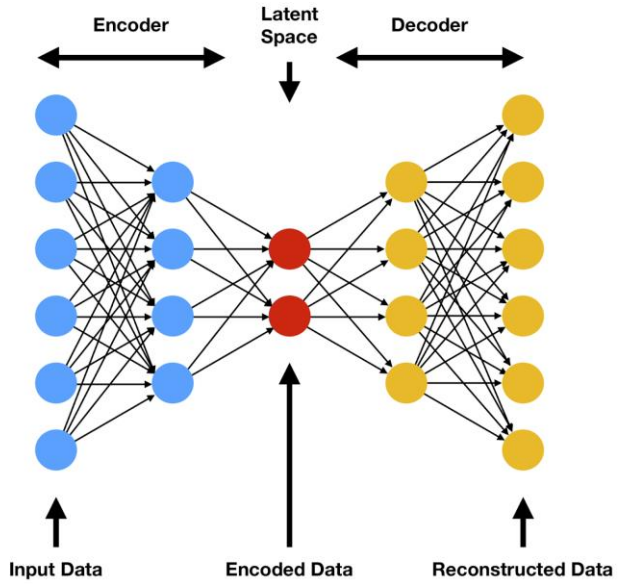
A kép illusztráció a konvolúciós hálóról Dr. Palágyi Kálmán prezentációjából

Egy olyan formája a konvolúciós hálónak, amelyet eredetileg objektumfelismerésre terveztek. Ez az architektúra a bemenő képek jellemzőit megtanulja (ahogy azt a korábban említett konvolúciós neurális háló tette) és ezeket a jellemzőket a tanulás végén csoportosítja ún. feature vektorokba. Ezek a vektorok a képen lévő objektumok jellemzőit reprezentálják, amit megtanult a modell. Alapvetően ezekkel a jellemzőket nem tudjuk megítélni, hogy pontosan milyen aspektust sikerült a hálónak megtanulnia, viszont ez már egy kellően absztrakt ötletet kínál ahhoz, hogy egy entitás független implementációt tudjak készíteni. Végül nem ezt az irányt választottam a végső megoldásom elkészítéséhez, mert találtam sokkal érdekesebb lehetőségeket is, amelyeket lentebb kifejtek.

2.8. Autoencoder

Habár kutatásaim alapján, ezt az architektúrát már nem használják sok helyen, mégis erre a számomra rendkívül érdekes speciális neurális hálóra esett a választásom. Az Autoencoder annyiban különbözik az előzőekben felsorolt hálózatoktól, hogy a bemenő adatok alapján megpróbálja tömöríteni a input rétegben lévő jellemzőket úgy, hogy a kimeneten végül elő tudja állítani ugyanazt a képet, amely eredetileg a bemenő rétegben jelent meg. Teszi ezt úgy, hogy a bemenő adatokat megpróbálja egy látens térben minél kisebb méretűre zsugorítani, azokkal a legfontosabb jellemzőkkel, amelyek egyértelműen leírják a bemenő kép milyenségét. A használata nagyon egyszerű, a Tensorflow pedig egy teljes modult biztosít a dokumentációja szerint, külön az Autoencoder-ek használatára kitérve. A rétegek zsugorítása nem kötelező, az architektúra tud működni abban az esetben is, ha csak egy látens dimenzió előállítása a cél, amely dimenziói teljesen megegyeznek a bemenő réteg dimenziójával.
A választásom végül egy eszelősnek tűnő ötlet működőképessége miatt esett erre az architektúrára. A Tensorflow dokumentációja leírja, hogy ez a háló képes zajszűrésre is,

amennyiben a kimeneti rétegben nem az eredeti kép előállítását várjuk el a modelltől, hanem a bemenő kép egy zajmentes változatát. Erre a tényre az az asszociáció jutott eszembe, , hogy ha képes zajszűrésre, akkor képes zaj készítésére is. Innen pedig csak egy gondolati lépcső volt az, hogy nem a dokumentációban említett természetes zajt kell elhelyezni a kimeneti rétegben, hanem a szegmentált objektumokkal rendelkező képet. Ennek a lehetőségét a mintaadatok segítségével azonnal kipróbáltam és mivel működőnek tűnt, ezért akkora sikerélménnyel tudtam erre az implementációra gondolni, hogy eldöntöttem, ebbe az irányba fogok elmenni a prototípus elkészítése során is. Későbbi kutatásaim, írásos formában is igazolták, hogy az Autoencoder nem csak képek tömörítésére, hanem képeken lévő objektumok szegmentálására és felismerésére is képes.
Az alábbi képen egy egyszerű Autoencoder felépítése látható. A bemeneti rétege úgy néz ki, mint egy egyszerű neurális háló, de a rejtett rétegek egyre szűkebb dimenzióval rendelkeznek. A legkisebb dimenziót nevezzük látens térnek (latent space), amely az encoder és a decoder között helyezkedik el. Az encoder feladata annyiból áll, hogy megtanulja tömöríteni a bemenő képet, és ebből létrehozzon egy látens teret. A decoder feladata pedig az, hogy ebből a látens dimenzióból újra előállítsa a kimenetet. Ezt szokás nevezni még felügyelet nélküli tanításnak is (unsupervised learning), ugyanis itt nincsenek címkék, amelyek súlyértékét meg kell tanulni, hanem a címkék helyett az input rétegben megjelenő adat jellemzőit szeretnénk megtanítani, ezért a címkék helyett a teljes bemenő adathalmaz dimenzióiban megjelenő súlyértékeket vizsgáljuk a kimeneten.



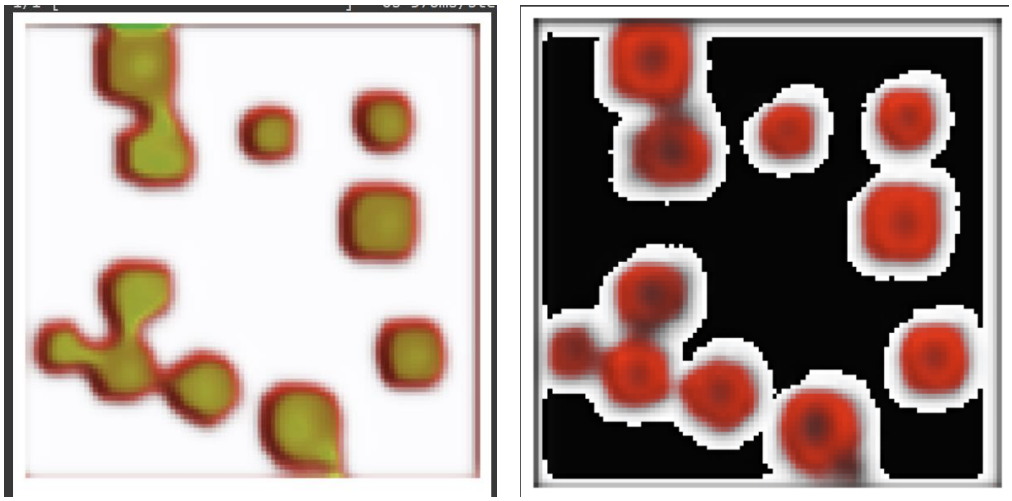
Autoencoder illusztráció Steven Flores bemutatójából a Comp Tree Inc. weboldalon

2.9. Egy pont megtanítása

Amikor úgy éreztem elég tudás birtokában vagyok ahhoz, hogy elkezdjem az implementációt, több konkrét ötlet is eszembe jutott a megvalósításhoz. Eddigi képfeldolgozási ismereteim arra engedtek következtetni, hogy ha HSV vagy HLS színmodellekkel dolgozom, akkor a modell a tanítása során könnyebben képes megjegyezni a képek fontos jellemzőit. Végül ezt a hipotézist nem volt időm megvizsgálni, de úgy gondolom pusztán RGB színmodell használatával több dologra kell figyelni a tanulás során, hiszen egyes képek esetén egy HSV modell esetén kieshetnek bizonyos értékek, amelyek nem adnak fontos reprezentációt, vagy előtérbe kerülhetnek. A prototípus esetén egy gyakorlati példával élve, a teszteléshez használt képeken, ahol a háttérkép teljesen fehér, ott például a Lightness értéknek csak 2 fontosabb értékkészlete lehet, hiszen egy képponton vagy megtalálható egy objektum, amely szignifikánsan érinti ezt az értéket, vagy nem, ez utóbbi esetben azonban fix értéket kell a kimeneten is megjeleníteni.

Egy hasonló ötletem azonban felmerült, mely során a bemenő képek értékkészletét nem módosítottam, de a kimeneten azt vártam, hogy 3 érték helyett (R, G, B) 4 értéket jelenítsen meg a modell. A negyedik érték egy indikátor volt, amely azt jelölte, hogy adott képpont mekkora valószínűséggel helyezkedik el egy objektum közepén. Több permutációban is kipróbáltam az bemenő és kimenő adatok reprezentálását, de úgy tűnt az Autoencoder modellem képtelen megtanulni az utolsó érték előállítását. Ezen a szinten úgy tűnt azt szerettem volna megtanítani a hálónak, hogy a bemenő, zajmentes képen jelenítsen meg egy redundáns zajt. Miután eltávolítottam a kimeneten a 4. paramétert, a modell szignifikánsan javulni kezdett. Ezen tapasztalatom alapján jelenleg azonban úgy látom, ha HSV vagy HLS tartományokat használtam volna, akkor sem jutottam volna előnyhöz, hiszen a modell szempontjából nem az az érdekes, hogy hol található egy objektum közepe, hanem az, hogy a bemenő adatok alapján milyen kimenő adatot kell előállítania, a látens térben lévő jellemzők megtanulását pedig befolyásolni vagy lehetetlen vagy túl nehéz.

Ez a hipotetikus felismerés azonban segített megérteni, hogy mit kell megjelenítenem a kimeneten, ahhoz, hogy az Autoencoder architektúra képes legyen megtanulni az objektumok pozícióját. Amint tudatosult bennem a viselkedési mechanizmus, átalakítottam a kimeneti képeket olyan formára, hogy ne csak az objektumok közepén legyen egy indikátor, hanem a teljes objektum egyszínű, szegmentált entitásként jelenjen meg. Az első eredmények így már teljesen egyértelművé tették számomra, hogy milyen folyamatok lehetnek egy konvolúciós háló mögött:



3. IMPLEMENTÁCIÓ

A szoftver megírása alapvetően két komponensre osztható szét. Az első komponens segítségével generáltam a tanításhoz és a teszteléshez adatokat, míg a második komponens maga az Autoencoder architektúra megírásából, annak tanításából és a modell elmentéséből állt. Minden kódot python nyelven írtam, és igyekeztem a python legfrissebb verzióját használni minden környezetben. Mivel ez volt az első MI projektem, ezért egy olyan megoldást kerestem a fejlesztői környezet kialakításához, amely segítségével könnyen és gyorsan tudok reagálni a hibákra és a rossz koncepciókra. Ez utóbbi esetben arra gondolok, hogy sokszor belefutottam azon problémákba, miszerint a teszt adatok generálása túl sokáig tartott, vagy a modell tanítása nagyon lassú volt. Az első modelleknél nem tudtam milyen arányokban kell gondolkodnom, így nem is tudtam eldönteni, hogy mennyi tanulóadatra van szükségem vagy, hogy mennyi ideig kellene, hogy fusson a modell tanítása, valamint a képadatok dimenziói és egyéb szükséges jellemzőinek milyenségéről sem tudtam megfelelő elképzelést alkotni.

3.1. Fejlesztői környezet összeállítása

Ahhoz, hogy kiküszöböljem ezen ismerethiányok által szabott akadályokat, készítettem egy lokális python környezetet is. Ez kezdetben csak arra volt jó, hogy egyszerű python kódokat tudjak futtatni a segítségével. Több gyakran használt python modult is telepítettem, amelyek az én munkámat is segítették, ilyenek voltak például a numpy, tensorflow vagy a multiprocessing modul. A modell tanítását azonban nem tudtam a lokális környezetemben

minden aspektusból megtenni, mert a tensorflow alapvetően CPU alapú erőforráskészlettel dolgozott, ami azt jelenti, hogy a tanításhoz szükséges alszálakat a processzor fizikai magjai között osztotta szét. Ez meglehetősen lassú tanítást eredményezett, így komplex architektúrával és sok adattal több órát vagy akár napokat is igénybe tudott venni egy-egy modell elkészítése.
Ezen probléma feloldása azonban viszonylag egyszerűnek tűnt. A Tensorflow dokumentációjában remekül le van írva, hogy milyen függőségeket kell telepíteni, hogy elérhetővé tegyük a GPU-t a tensorflow számára. Mielőtt leírnám az ötleteimet, szeretném kiemelni, hogy multiboot rendszerben azért nem gondolkodtam, mert a notebookom Windows kompatibilis és nem készültek megfelelő driverek hozzá, hogy unix környezettel is működni tudjon. Anélkül, hogy belemennék a részletekbe, a Tensorflow 2.10 natív módon is támogatta a Windows környezetet, azonban egy sebezhető komponens nyugalmazása után Windows 10-től felfelé már csak WSL2 segítségével lehet kialakítani egy lokális Tensorflow környezetet GPU használattal. Mivel ehhez a területhez keveset értek, ezért arra gondoltam, hogy készítek egy virtuális gépet egy Linux disztribúcióval, majd erre telepítek minden függőséget és a virtuális környezetben folytatom az implementációt. Hamar rá kellett jönnöm, hogy a virtuális gép és a host között a GPU megosztása nem egyszerű feladat, ráadásul a Microsoft ezen funkció natív támogatását már évekkel ezelőtt törölte a HyperV rendszeréből. Ezek után elgondolkodtam azon, hogy elinduljak-e egy olyan irányba, hogy ezt a problémát megpróbálom feloldani egy másik virtuális környezet összeállításával, de sajnos csak a VMWare által nyújtott lehetőségekre találtam rá, amely ígéretesnek tűnt, de mivel nem volt meg a megfelelő tudásom, ezért nem szerettem volna jelenleg ebbe az irányba elmenni.
Ezek után tettem még egy próbát a lokális környezet elkészítését illetően WSL2 segítségével. Feltelepítettem az nVidia Studio drivert, az Anaconda package managert, CUDA toolkit-et és a cudNN SDK-t. A telepítése ezeknek a függőségeknek nem volt nehéz, és sikeresen tudtam minden komponenst külön tesztelni. Azonban a Tensorflow továbbra is a CPU erőforrásait használta és sajnos nem tudtam rájönni, hogy miért, ezért úgy döntöttem, hogy nem szeretnék egy prototípus implementálásához elkészíteni egy ilyen környezetet, lévén biztosan van egyszerűbb módja is, ami ráadásul hordozhatóvá is teszi a kódot. Ez utóbbi alatt nem a verziókövetésre gondolok, hanem arra, hogy célom volt az, hogy a modellt több számítógépen is képes legyen lefuttatni, mert a mindennapjaim során több eszközről is bejelentkezem és vannak esetek, amikor az otthoni számítógépemhez nem férem hozzá.

Végül annál a megoldásnál maradtam, amit korábban már említettem az előző fejezetekben is, létrehoztam egy egyszerű python környezetet, amely segítségével képes voltam generálni teszt adatokat és modell tanítást futtatni, de csak a CPU erőforrásaival.

3.2. Adatgenerálás

Mielőtt belemennék az adatgenerálás implementációjának a leírásába, szeretném újra felidézni az első ötleteket. Az alapkoncepcióm az volt, hogy egy egyszerű kamerával készített képekből készítek több halmazt a tanításhoz, a teszteléshez és a validáláshoz. Azonban amint elkezdtem a kódot írni valamiért bennem volt, hogy ezekből a képekből több ezerre is szükségem lehet, hiszen csak így tudok egy modellt megfelelően megtanítani a látens jellemzők feltérképezésére. Egy mobiltelefon használatával azonban sokszáz kép elkészítése is időigényes lehet, ezért utóbb már inkább a mintaképek generálásában gondolkodtam. Adatgenerálás címszó alatt azt értem, hogy szükségem volt egy olyan algoritmusra, amely segítségével a modell tanításához szükséges bemenő adatokat könnyedén el tudom készíteni. Korábban már írtam az input struktúráját érintő koncepciókról, itt most kifejteném a különböző koncepciók struktúrális megvalósítását.
Első ötletem az volt, hogy elkészítek egy fehér háttérrel rendelkező képet 1000x1000 képpontos méretben. A képre ezután ráhelyezek véletlenszerű pozícióban minimum n darab és maximum m darab kisebb képet, amely a háttérképen lévő objektumot hivatott reprezentálni. Hamar rájöttem, hogy a kép mérete nem lesz megfelelő, de erről egy későbbi fejezetben írnék bővebben. A mintaképeket generáló algoritmust hamar megírtam, először csak pár véletlenszerű objektumot helyeztem el a háttérképen, később azonban kicseréltem őket napraforgókra.

Ez azonban nem volt megfelelő formátumban ahhoz, hogy egy neuronhálónak át tudjam adni az input rétegben. Ahhoz, hogy ezt meg tudjam tenni, értelemszerűen olyan adatstruktúrát

kellett kialakítanom, mely számokkal írja le a képen lévő információkat. Habár volt pár ötletem ennek a feladatnak a megoldására is, úgy láttam jónak, hogy először megnézem a dokumentációt, és analizálom, hogyan használják az Autoencoder mintakódjában a képadatok átadását az input rétegben.
A példakódban szürkeárnyalatos képekkel dolgozik a modell. Ez számomra abból a szempontból jelentett egy kisebb kihívást, hogy nekem, a már korábban említett HSV színpalettával kellett dolgoznom. Nem volt nehéz azonban rájönni, hogy a bemenő képeket egy 3 dimenziós tömbbel reprezentálták. Az első dimenzióban a tanításhoz szükséges képeket, a második dimenzióban a képek sorait, a harmadikban pedig a képek 1 és 0 közé szorított, bemenő súlyként értelmezhető, szürkeárnyalatos pixelek átlagát tárolják. Ahhoz, hogy egy HSV, vagy egy RGB színreprezentációt tudjunk ebben a struktúrában őrizni, egy újabb dimenzió bevezetésére volt szükség.
Habár ezzel elértünk ahhoz a ponthoz, ahol megoldódott a bemenő adatok átadása a modell számára, még egy problémával szembe kellett néznem. A nagy mennyiségű adathalmazzal való tanítás igen erőforrásigényes művelet lehet, és ez a jelenség számomra is nehézségeket okozott. Az első próbálkozások az 1000x1000 képpontnyi méretű képekkel túlzott mértékű memóriahasználatot eredményeztek a tanítás során. Egyértelmű volt számomra, hogy nem tudok több, mint 200 GB-nyi adatot egyszerre betölteni a memóriába, értelmetlen is lett volna. Ezért az egyetemen tanultak alapján azon gondolkodtam, hogy hogyan tudom kisebb halmazokra, ún. batchekre szeparálni a bemenő adatokat. A Tensorflow erre biztosít saját megoldásokat, amelyek segítségével DataSet-en keresztül biztosítja az adatok felolvasását tanítási idő alatt. Én a TFRecordDataSet nevű python osztályt használtam, ezzel tulajdonképpen elég volt megadni a TensorFlow számára, hogy a képek hol találhatók meg, milyen útvonalon és ezek után az implementáció a modell tanításakor ebből a DataSetből egyenként töltötte be és használta fel a fájlokat. Ennek a megoldásnak az implementálása nem volt teljesen egyértelmű számomra, így több dolgot is kipróbáltam előtte, de végül az alábbi kód segítségével sikerült elérnem ezt a fajta adatbetöltési mechanikát.
Az alábbi kód a bemenő paraméterében egy könyvtárat vár. Ebben a könyvtárban végigmegy az összes fájlon és betölti azokat egy python tömbbe. Majd a végén visszatér ezzel a tömbbel a klienshez.

```
def get_file_paths_by_directory(directory_path):
  file_paths = []
  for root, _, files in os.walk(directory_path):
    for file in files:
      file_paths.append(os.path.join(root, file))
  return file_paths
```

A DataSet inicializálása a következőképp történik:

```
train_dataset = tf.data.TFRecordDataset(train_file_paths,
compression_type='GZIP')
    train_dataset = train_dataset.map(decode_image_from_tfrecord)
```

Ebből a kódból remekül látszik, hogy a képek gzip tömörítéssel vannak tárolva, a minél kisebb helyigény elérése érdekében.
Mielőtt feldolgozásra kerülnek a tanuló algoritmus belsejében, ki kell a fájlokat csomagolni, és normalizálni kell az adatot. Ez utóbbit maga a TensorFlow automatikusan megteszi, ha megadjuk a compression_type argumentumot neki. A normalizálást pedig a fent említett map metódus használatával tudjuk biztosítani oly módon, hogy egy delegált metódust regisztrálunk a DataSethez, amely lefut mielőtt az adatok ténylegesen elkerülnének az input neuron rétegbe.

```
def decode_image_from_tfrecord(example_proto):
  feature_description = {
    'input': tf.io.FixedLenFeature([CONSTANTS["IMAGE_WIDTH"] *
CONSTANTS["IMAGE_HEIGHT"] * CONSTANTS["NUMBER_OF_ATTRIBUTES"]], tf.float32),
    'output': tf.io.FixedLenFeature([CONSTANTS["IMAGE_WIDTH"] *
CONSTANTS["IMAGE_HEIGHT"] * CONSTANTS["NUMBER_OF_ATTRIBUTES"]], tf.float32)
  }
  example = tf.io.parse_single_example(example_proto, feature_description)
  decoded_input = tf.reshape(example['input'], (CONSTANTS["IMAGE_WIDTH"],
CONSTANTS["IMAGE_HEIGHT"], CONSTANTS["NUMBER_OF_ATTRIBUTES"]))
  decoded_output = tf.reshape(example['output'], (CONSTANTS["IMAGE_WIDTH"],
CONSTANTS["IMAGE_HEIGHT"], CONSTANTS["NUMBER_OF_ATTRIBUTES"]))
  if (CONSTANTS["IS_SEGMENTATION_ENABLED"]):
    return [ decoded_input ], [ decoded_output ]
  else:
    return [ decoded_input ], [ decoded_input ]
```

A kódrészletben az látható, ahogy az ún. TFRecord fájlok feldolgozásra kerülnek. Ezek a fájlok lényegében bináris szekvenciákkal tárolják a bennük lévő adatot. A szekvenciák struktúráját a feature_description nevű változó írja le egy kulcs-érték pár segítségével, ez

alapján tudjuk az ún. example adattípusból kiolvasni az értékeket. A kódban a kiolvasás után normalizáljuk az adatot és a végén egy debug-flag segítségével beállítható az, hogy az Autoencoder az input rétegben lévő képet tanulja meg az output rétegben előállítani, vagy pedig szegmentálni tanuljon meg.

3.3. Erőforrásigény problémáinak megoldása

Részben már kitértem erre a témára, mint kezdő, mesterséges intelligenciával foglalkozó amatőr, arról szeretnék részletesebben írni, hogy még nem tudtam előre megjósolni, hogy milyen mennyiségű adatra van szükségem, valamint, hogy az adatok milyen részletességűek és struktúrájúak legyenek. Ez az első lépéseknél nagyon lassú haladást eredményezett, hiszen vagy túl sokáig tartott a modell tanítása vagy túl sok idő volt tesztadatot generálni. Ez utóbbi kiküszöbölésére először apró lépésekkel segítettem a saját haladásomat, például többszálúsítottam a generáló logikát és beépítettem debug lehetőségeket, amelyek különböző formában jelenítették meg a kimenetet, így gyorsan és hatékonyan tudtam ellenőrizni, hogy a generált adathalmaz jó formátumban került előállításra.
Miután sikerült egy a változtatásokra viszonylag adaptív kódot megírnom a generáláshoz és nyomon tudtam követni a kimenet változását, gyorsan kiderült, hogy a problémát a nagy méretű, 1000x1000 pixel méretű képek okozzák. Ezért mindenképp jónak láttam azt, hogy kipróbáljam, meddig tudom csökkenteni a méretet anélkül, hogy komolyabb minőségbeli romlást véljek felfedezni a prototípuson. Sajnos mivel nem volt még kész modell architektúra és nem volt meg a megfelelő tudásom sem az MI-t illetően, ezért nem tudtam mihez kötni a méret változtatásokat, így csak egy hipotetikus módszert alkalmazva, találomra elkezdtem kisebb méretű képeket generálni. A végén rájöttem, hogy akár egy 50x50 pixel méretű kép is elég lehet a céljaim eléréséhez, de ezt a méretet azért nem tartottam jónak, mert a Python alapkönyvtárai nem tudták megfelelő minőségben skálázni a képen lévő objektumok méretét, így sokszor elmosódást vagy torzulást tapasztaltam az entitásokon. Azt még nem említettem, de a Python nyelvvel is ebben a projektben találkoztam mélyebben először, így mondanom sem kell, nem tartottam jó ötletnek - főleg a határidőket és a sikerélmény progresszív kibontakozását figyelembe véve -, hogy komplexebb külső könyvtárakat ismerjek meg és alkalmazzak a prototípus implementálásához. A képek méretét illetően a végén úgy döntöttem, hogy 100x100-as képekkel dolgozom, amelyen 8-12 darab objektumot helyezek el, a háttérkép nagyságához illeszkedő relatív méretekkel. Miután sikerült az első működő modellt betanítanom, próbálkoztam még a 256x256-os képmérettel, amely valószínűleg túl

aránytalan növelés volt, amivel a prototípus architektúrája már nem tudott megbirkózni. Ezért a számolások megkönnyítése és az architektúra sértetlenségének megőrzése céljából a 128x128-as bemeneti képméretet választottam a végső implementációban. Ez a méret kellőképpen kevés erőforrást igényelt, valamint a modell tanítási idejét is szignifikánsan lecsökkentette.
Habár a képméret csökkentése megoldotta a tanítás során felmerült problémák java részét, egy aspektust még nem szabad figyelmen kívül hagyni. Mivel a bemenő adatok már jóval kisebb térfogattal rendelkeztek, ezért nem volt már szükség jelentős mennyiségű tesztadatra a tanuláshoz. Ez azt jelenti, hogy az első koncepciókban meghatározott 10000 darab tesztkép már túl soknak bizonyult, hiszen a prototipizált környezet egyszerűsége miatt a modell elkezdett túltanulni és specializálódni. Az utolsó modelleknél már elég volt 30 darab képet használni a tanuláshoz és 10 darab képet használni a teszteléshez. Ez a szám jól látható, hogy jelentősen degradált az első próbálkozásaimhoz képest, amely azt is jelenti, hogy a tanulási idő is egyenes arányban csökkent.
Ezekkel a jellemzőkkel már nagyon gyorsan tudtam dolgozni és hamar tudtam reagálni a hibás architektúrára, valamint ahogy egyre jobban beleástam magamat a mesterséges intelligencia érdekes világába, egyre több logikai összefüggés is összeért a fejemben, amely segítségével racionálisnak tűnő apró módosításokkal az architektúrán, kimagaslóan jobb végeredményt tudtam elérni.

3.4. Utófeldolgozás

Habár, kutatásaim szerint, magát az Autoencodert meg lehet tanítani arra, hogy olyan kimenetet állítson elő, amely már önmagában interpretálható, egy algoritmizálható utófeldolgozás azonban mégis egyszerűbbnek tűnt számomra. Egyrészt azért döntöttem így, mert sok helyen azzal az elvvel találkoztam, mely szerint az utófeldolgozás egy szerves része kell legyen a képfeldolgozó algoritmusoknak, másrészt pedig azért, mert habár az Autoencoder látens rétegeit ki tudtam volna használni a megfelelő információk előállítására, de sajnos még nem rendelkezem a szükséges mesterséges intelligencia megfelelő alkalmazásához szükséges gyakorlati és elméleti tudással, így a látens rétegben, ha sikerült is valaha előállítanom egy konkluzív állapotot, nem tudtam az abban lévő információkat megfelelően kibányászni.
Azonban a kimeneti rétegben sikerült közel 95 százalékos pontossággal előállítanom egy képet, amelyen szegmentált objektumok találhatók. Ez az eredmény már elegendő volt ahhoz,

hogy meg tudjam számolni a felismert entitásokat, még akkor is, ha az részben hibás kapcsolódási pontokat tartalmazott. Ahhoz, hogy a számlálást implementálni tudjam, először normalizálni kellett a kimeneti rétegben lévő adatot. Egy egyszerű példával élve illusztrálom az egész folyamatot az alábbiakban.

Bemenő kép

Szegmentált kép

Normalizált adathalmaz (részlet)

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

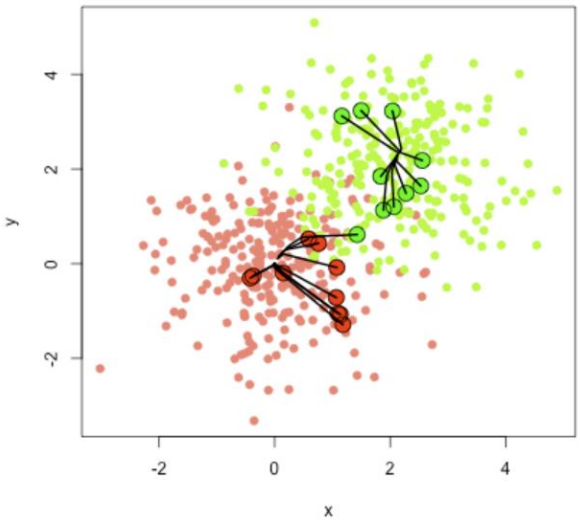
3.5. Objektumszámlálás DBSCAN segítségével

Az előző fejezetből tehát a következő kimenetet kaptuk:

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

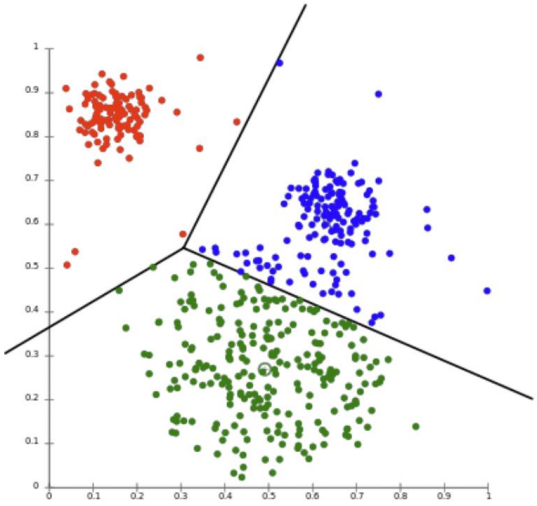
Ha megvizsgáljuk ezt a tömböt, látható, hogy leginkább egy bináris képre hasonlít, hiszen egy objektumot vélhetünk felfedezni a közepén. Ez azért jó, mert így egy nagyon egyszerű implementációval elegendő azon képpontok összegyűjtése, amelyek homogén egységet alkotnak. Erre több módszert is találtam, sokat közülük a TensorFlow is támogat beépített komponensekkel, de mivel MI témában szerettem volna maradni, ezért olyan algoritmusokat kerestem, amelyekkel bővíthetem a tudásom ebben a kontextusban. Ebben a fejezetben nem szeretnék belemenni nagyon a részletekbe, ezért csak a legfontosabb algoritmusokat említeném meg, amelyeket kicsit részletesebben is megismertem, mielőtt eldöntöttem melyikkel végezném el az utolsó feladatrészt. Ezek a DBSCAN, a Mean-Shift klaszterezés és a K-Means klaszterezés voltak.

A Mean-Shift klaszterezés egy számomra nagyon érdekes módszeren alapul. Ez egy olyan nemparametrikus osztályozás, amely az adatokat egy sűrűségfüggvény segítségével keresi meg azáltal, hogy egy csúcspontba próbálja őket csoportosítani. Ezt a módszert végül elvetettem, mert kutatásaim alapján a Tensorflow nem biztosít beépített komponenst ehhez az algoritmushoz.



Mean-Shift illusztráció az R-Blogger oldaláról

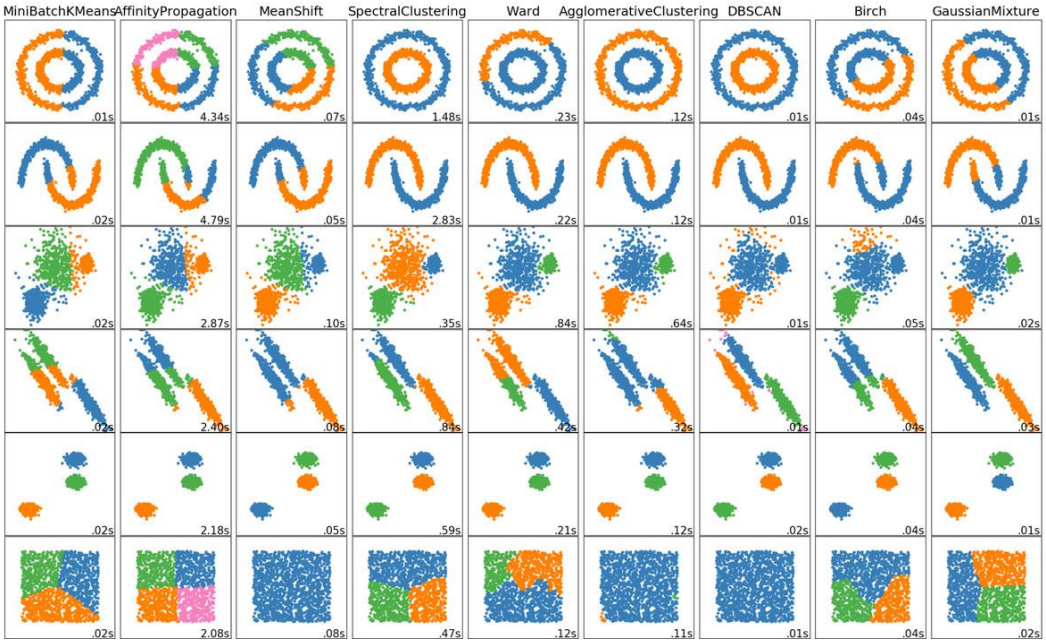
A K-Means algoritmus véletlenszerű, n darab ún. klaszterközépponthoz rendel adatokat és ennek folyamatos újraszámolásával keresi meg a klaszterhatárokat. Sajnos ahhoz, hogy ezt a módszert tudjam használni szükség volt tudni a klaszterek pontos számát, ami tulajdonképpen a képen lévő objektumok darabszámát reprezentálta volna, ezért ezt a módszert nem tudtam alkalmazni.



K-Means illusztrációja az AWS oldaláról

Végül a választásom a DBSCAN nevű algoritmusra esett, amely szomszédsági viszonyok alapján térképezi fel az egy klaszterbe tartozó képpontokat. Ennek használata nagyon könnyű volt és a kimeneti rétegben lévő adathalmaz - normalizálás után természetesen -, teljesen

elfogadható bemenő adatként szolgált a DBSCAN számára, de igazából ez a módszer valamelyest rugalmas a bemenő adat milyenségét illetően.



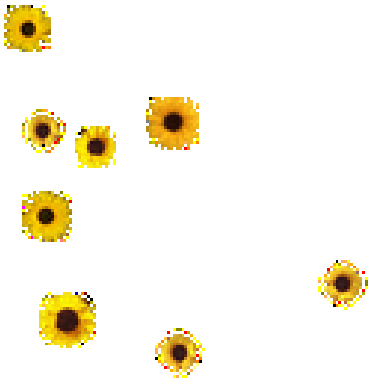
A DBSCAN kimeneteinek illusztrációja a MACHINE LEARNING EXPLAINED oldalról

A DBSCAN implementációja a következő módon történt:

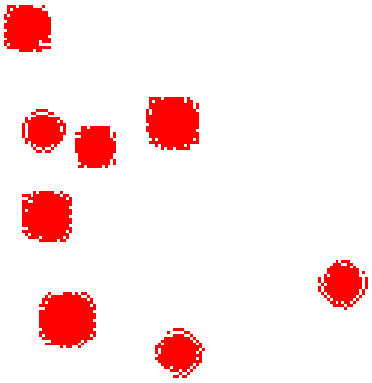
```
indicated_images = []
for image in images:
  rows = []
  for row in image:
    pixels = []
    for pixel in row:
      r, g, b = pixel
      if (r > 239 and g < 11 and b < 11):
        pixels.append(1)
      else:
        pixels.append(0)
    rows.append(pixels)
  indicated_images.append(rows)

first_image = indicated_images[0]
eps = 2
min_samples = 5
dbscan = DBSCAN(eps=eps, min_samples=min_samples)
labels = dbscan.fit_predict(first_image)
unique_labels = np.unique(labels)
print(f"Number of detected objects: {len(unique_labels)}")
```

Ebben a kódban az images változó az Autoencoder által prediktált képek tömbjét tartalmazza. Ezen a tömbön végigiteráltam és minden pixel képpontjait megvizsgálva egy új, bináris képet készítettem. A bináris képre azért volt szükség, mert így a DBSCAN algoritmus sokkal könnyebben tudta megtalálni a klaszereket. A bináris kép elkészítésére azért volt lehetőség, mert az Autoencodert nem az eredeti kép előállítására tanítottam meg, hanem szegmentálásra, amely a következőképpen néz ki:

Eredeti kép



Szegmentált kép

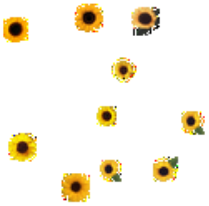


Ez a kimenet már közel egy bináris képnek felel meg, de ha részletesen megnézzük a különböző képpontok bár nem szignifikánsan, de eltérnek. Ennek kiküszöbölésére alakítottam át ezt a szegmentált képet a már korábban említett, 1-eseket és 0-ásokat tartalmazó bináris képpé. Miután ez megtörtént, a teszt kedvéért kiválasztom az első képet - habár igazából bármelyik képet kiválaszthatom -, és egy DBSCAN példányon meghívom a fit_predict metódust, amely felállítja a predikciókat a képen. Ahhoz, hogy ebből a predikcióból megkapjuk az objektumok számát, meg kell számolni az egyedi címkéket a dbscan unique metódusával.
Ezzel a módszerrel, a kódban látható előre definiált metaparaméterek segítségével - melyek tapasztalataim szerint a legjobbnak bizonyultak -, a következő eredményt értem el:

Felismert objektumok: 8

Felismert objektumok: 12

Felismert objektumok: 9



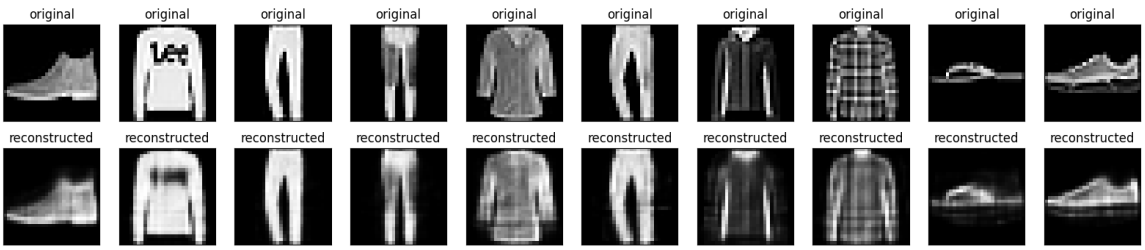
Felismert objektumok: 9

Felismert objektumok: 9

Látható, hogy a legtöbb esetben majdnem minden objektumot megtalál az algoritmus, azonban azokon a helyeken, ahol az entitások nagyon szorosan egymás mellé vannak pozícionálva, azokat egy objektumként kezeli a modell. Habár ezt a gyermekbetegséget nem tudtam a prototípuson javítani, próbálkoztam nagyobb méretű képekkel is. Úgy tűnik a kép méretének növekedésével, és a modell architektúra komplexitásának inkrementálásával a kimeneti képek egyre élesebbé válnak. Erre példát sajnos nem tudok mutatni, mert minél komplexebb a modell a tanulási idő annál nagyobb, viszont egy másik aspektust még fontos figyelembe venni.
A DBSCAN, ahogy azt korábban említettem, szomszédsági viszonyokat figyel. Ezzel az a probléma, hogy a természetben nagyon nehéz olyan képeket készíteni, ahol az objektumok nem lógnak egymásra. Ha a szegmentálás még működik is, az objektumok megszámlálása már nehezebb. Ennek kiküszöbölésére a prototípus koncepciójában nem kívántam próbát tenni, de léteznek olyan módszerek amelyek segíthetnek a határok éles meghúzásában, mint például a szegmentáló modell újratanítása oly módon, hogy egy szín helyett különböző színeket használjon a különböző objektumokhoz vagy más típusú algoritmus használata a DBSCAN helyett.

3.6. Autoencoder architektúra

Elérkeztünk a prototípus magjához, a neurális háló architektúrájához. Habár gyakorlatilag ezzel a résszel töltöttem el a legtöbb időt úgy kutatásban, mint implementációban, mégis úgy érzem, hogy szinte semmit nem tudok még a neurális hálókról. Ennek oka talán az, hogy egy neuronháló teljeskörű megértése igen komplex és sokszor nehezen kiszámítható feladat. Az első próbálkozásaim a TensorFlow által kínált Autoencoder mintakódból ihlettek, amely az ún. Fashion MNSIT adatkészletet felhasználva mutatta be a háló képességeit.



Autoencoder kimenetének illusztrációja a TensorFlow mintakódjából

Ebből a mintából tanultam meg, hogy hogyan kell felépíteni egy alapvető architektúrát, milyen bemenő adatokat kell megadnom és milyen kimenetet kapok a hálózat kimeneti rétegében. Természetesen sejthető volt, hogy ez a példakód nem lesz alkalmas az én céljaim megvalósítására, ezért igyekeztem a következtetéseket levonni és megérteni a mögötte lévő koncepciót.
Az első probléma amivel szembesültem az az volt, hogy a bemenő adatok ebben a mintakódban szürkeárnyalatos objektumok. Ez azt jelenti, hogy nekem át kellett alakítanom az adatok struktúráját arra, hogy képes legyen 3 csatornát fogadni a modell. Ezt sikerült úgy megtennem, hogy bővítettem 1 dimenzióval az eredeti tömböt, így egy egyszerű példán keresztül bemutatva a következő struktúrát kaptam:

```
[[[255., 255., 255.], [255., 255., 255.], [255., 255., 255.]],
 [[255., 255., 255.], [255.,   0.,   0.], [255., 255., 255.]],
 [[255., 255., 255.], [255.,   0.,   0.], [255.,   0.,   0.]],
 [[255.,   0.,   0.], [255., 255., 255.], [255.,   0.,   0.]],
 [[255., 255., 255.], [255.,   0.,   0.], [255.,   0.,   0.]],
 [[255.,   0.,   0.], [255.,   0.,   0.], [255.,   0.,   0.]]]
```

Ebből a példából jól kivehető - amely jelenleg a szegmentált kép képpontjait tartalmazza, ezért figyelhető meg csak 2 színű pixel rajta -, hogy hogyan kerül reprezentálásra a 3 csatorna és a kép oszlopainak és sorainak megfeleltetése. Az első dimenzió maga a kép, a második dimenzió tartalmazza egy sor leírását a harmadik dimenzió pedig a sorban lévő pixelek 3 csatornás RGB leírása.
Ezt a tömböt a korábban említett módon tfrecord.gz fájlokba csomagoltam, úgy, hogy a tfrecordban az eredeti kép és a szegmentált kép is helyet kapjon a megfelelő feature-ök leírásával.

Elérkeztünk magához a modell architektúrához, amely a következőképp néz ki:

```
class Autoencoder(Model):
  def __init__(self, latent_dimension):
    super(Autoencoder, self).__init__()
    self.latent_dimension = 12
    self.kernel_size = 3

    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(128, 128, 3), name="encoder_start"),
      layers.Conv2D(75, (3, 3), activation='relu', padding='same', strides=1,
name="layer_10"),
      layers.Conv2D(50, (3, 3), activation='relu', padding='same', strides=1,
name="layer_20"),
      layers.Conv2D(25, (3, 3), activation='sigmoid', padding='same',
strides=1, name="layer_30"),
    ])
    self.encoder.summary()
    self.decoder = tf.keras.Sequential([
      layers.Input(shape=(128, 128, 25), name="decoder_start"),
      layers.Conv2DTranspose(25, kernel_size=3, strides=1, activation='relu',
padding='same', name="layer_40"),
      layers.Conv2DTranspose(50, kernel_size=3, strides=1, activation='relu',
padding='same', name="layer_50"),
      layers.Conv2DTranspose(75, kernel_size=3, strides=1, activation='relu',
padding='same', name="layer_60"),
      layers.Conv2D(3, kernel_size=(3, 3), activation='sigmoid',
padding='same', name="layer_70")
    ])
    self.decoder.build(input_shape=(1, 128, 128, 3))
    self.decoder.summary()

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded
```

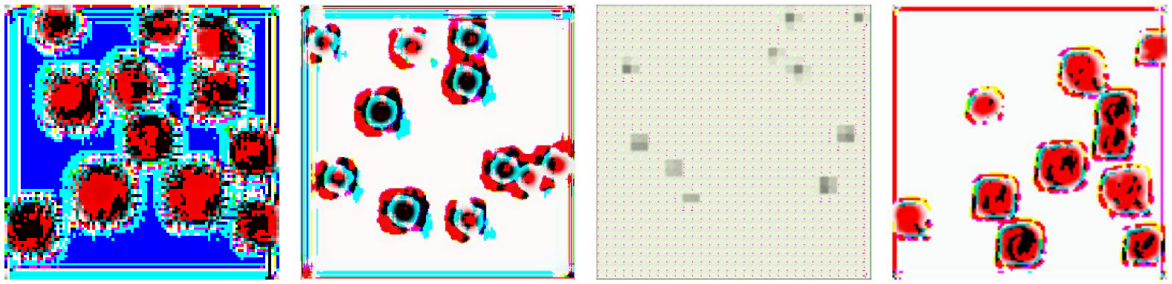
Ahhoz, hogy könnyebben megérthessem az architektúra felépítését, úgy döntöttem kiszervezem egy külön python osztályba az egészet, oly módon, hogy az eredeti TensorFlow által biztosított Model osztályt felülírom és kiegészítem a saját projektem aspektusaimmal. Az első metódus, melyet felülírtam az maga az __init__ függvény, ahol természetesen az inicializáláshoz kapcsolódó lépéseket implementáltam. Ez jelen esetben a korábban említett látens dimenziók számának és a kernel méretének megadása, majd ezt követően az encoder és a decoder architektúrájának definiálása. A .summary() metódus segítségével azt is ellenőrizni

tudtam, hogy a tényleges modellarchitektúra hogyan néz ki, és segítségével könnyen tudtam reagálni a hibákra, ha esetleg változtatni kellett rajta valamit. Ahhoz, hogy könnyebben el tudjam magyarázni a részleteket, itt látható a summary metódus végeredménye:

3.6.1. Encoder architektúra

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer_10 (Conv2D)           (None, 128, 128, 75)      2100

 layer_20 (Conv2D)           (None, 128, 128, 50)      33800

 layer_30 (Conv2D)           (None, 128, 128, 25)      11275

=================================================================
Total params: 47175 (184.28 KB)
Trainable params: 47175 (184.28 KB)
Non-trainable params: 0 (0.00 Byte)
```

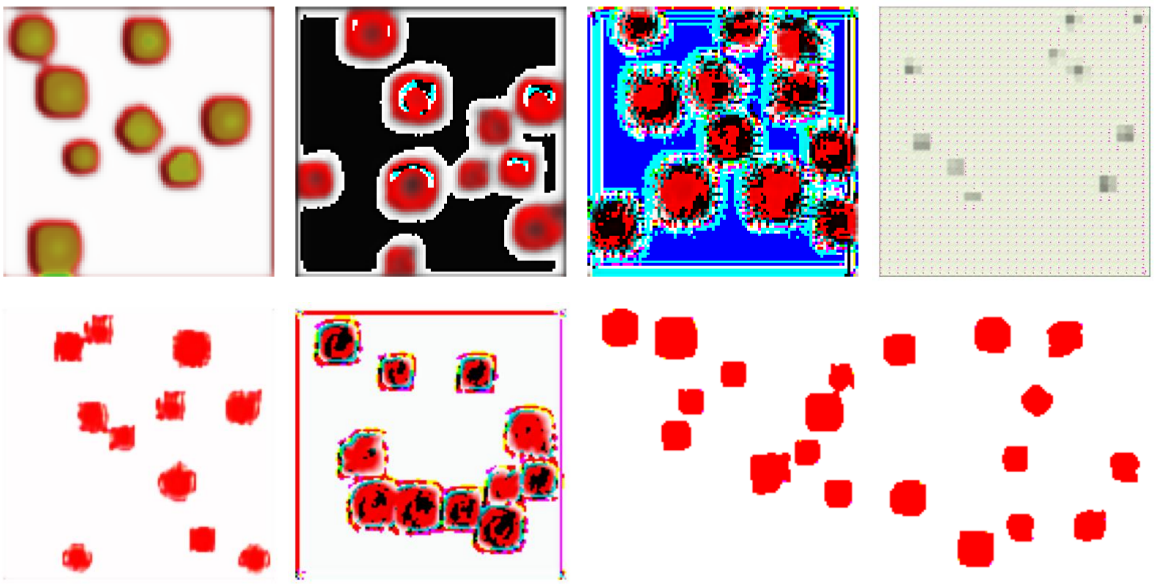
Az Encoder egy olyan szekvenciális architektúrát kapott, amely 3 darab konvolúciós rétegből áll. Erre azért volt szükség, mert fontos volt a képpontok közötti kapcsolat megőrzése, hiszen egy egyszerű “fully-connected” rétegekből álló háló nem tudta volna megtanulni az összefüggéseket a kép elemei között. Habár próbálkoztam azzal is, hogy egyszerű “Dense” rétegeket veszek fel, - amely a TensorFlow által biztosított fully-connected réteg implementációja -, de ennél a projektnél, ilyen kis méretű képeknél ez a módszer nem hozott nagyobb profitot a tanulás során, sőt inkább rontott az eredményen és a modell elkezdett hibás következtetéseket levonni. A következő néhány minta erre:



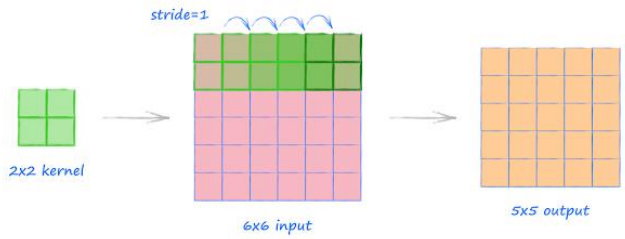
Ezen végeredményeket látva igyekeztem megérteni több-kevesebb sikerrel, hogy mi az, amit nem értek és milyen változtatást kellene a rendszerben eszközölnöm ahhoz, hogy a megfelelő

eredményt kapjam meg. Végül a gyakorlati tapasztalataim arra engedtek következtetni, hogy kövessem az Occam-Borotva elvét és igyekezzek minél egyszerűbb, de még szükségszerűen komplex architektúrát kialakítani. Sajnos sok miértre még nem találtam meg a választ, de fontos tapasztalatokkal gazdagodtam, amelyek segítettek a mesterséges intelligencia hatalmas világában magabiztosabb tájékozódást nyújtani.

Egy nagyon fontos aspektus például az volt, hogy a bemenő képek méretét érdemes úgy megválasztani, hogy azok a dimenziók a megfelelő rejtett rétegekben maradék nélkül összevonhatók (pooling lépés) legyenek. Ezért választottam ki a kép méretének a 128x128-as felbontást, ezzel könnyedén tudtam csökkenteni a méretét a képnek egyre jobban és jobban a látens térben. Mint kiderült, erre azonban az én prototípus képeim nem reagáltak túl jól, ez csak akkor működött valamennyire, de nem szignifikáns mértékben, ha a képek méretét megdupláztam, tehát 256x256-os felbontást alkalmaztam. Ezért az Autoencoder architektúrát úgy választottam meg, hogy a kép mérete az eredeti maradjon a látens térben is, de magát a filterek számát - amely alapesetben az adatok dimenzionalitásának a csökkentésére szolgál - lecsökkentettem 25-re. Ezzel a próbálkozással egyre közelebb és közelebb kerültem a megoldáshoz és, ha egy nagyobb amplitúdót tapasztaltam a végeredményt illetően, akkor változáshoz kapcsolható paramétereket visszaírtam az eredeti értékére. Az alábbi ábrán sorba rendeztem a tapasztalt végeredmények fontosabb kilengéseit, a historikusságot is figyelembe véve - a képek nem ugyanabból a mintából készültek, hiszen alkalmanként új adatokat kellett generálnom, ha szükség volt változtatni az adatstruktúrán:



Sajnos a legelső mintákból már nem tudtam félretenni, mert a kutatásom során nem mindig tudtam figyelni a verziókövetésre, hiszen igyekeztem használható végeredményt alkotni, azonban az első kép az már egy olyan architektúrát kapott, amely kellően minimális és képes megtanulni az objektumok szegmentálását, még akkor is, ha nem teljesen helyes módon.
A képek sorbarendezésével arra kívánom felhívni a figyelmet, hogy az architektúra apró módosítása milyen nagy mértékű változásokat eszközölnek. Az felső képsor még nagyon komplex és kezdetleges architektúrát tartalmazott egyszerű Dense rétegekkel is. Amint ezen rejtett rétegektől megszabadultam az alsó eredmény tárul a szemem elé. Itt már az vehető észre, hogy az apró módosításaim akkor sem kapnak akkora kilengést, ha valamit nagyon rosszul csináltam. A legjobb eredmény, amelyet el tudtam érni az az utolsó képen látható, itt a tanulás végén a veszteségfüggvény értéke 0.0108 és 0.0084 között ingadozott különböző tanulási ciklusokban.
A meg nem említett hiperparaméterek az architektúrában, mint például az aktivációs függvény, a padding és a strides, ugyanúgy tapasztalati tényezők alapján kerültek definiálásra. Egyes eseteket értettem, hogy miért van szükség például a relu aktivációs függvényre, de azt például nem tudtam elméletben is megérteni, hogy a sigmoid aktivációra miért van szükség az utolsó Encoder rétegben. A strides és a padding egyértelmű volt számomra, ezek a pooling művelet során a számításokban fontos szerepet játszottak, így tudtam megfelelően léptékkel csökkenteni a látens dimenziót a rejtett rétegeken keresztül. A padding paraméter “same” értéke például azt befolyásolta, hogy a bemenő és a kimenő réteg ugyanakkora dimenziókkal fog rendelkezni. A strides paraméter pedig a lépésközt definiálja, amely a kernelméret függvényében határozza meg a kimenetet. Ezeket pedig mint korábban említettem, főként azért választottam meg a kódban látható értékre (padding = “same”, strides = 1), mert a képméret csökkentése a rétegek között nem hozta meg a várt eredményt. Arra azonban nem találtam matematikailag is levezethető választ, hogy ez a tapasztalat miért jelenik meg ennél a prototípusnál.



Kernel és stride illusztrációja MYO NeuralNet bemutatójából

3.6.2. Decoder architektúra

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer_40 (Conv2DTranspose)  (None, 128, 128, 25)      5650

 layer_50 (Conv2DTranspose)  (None, 128, 128, 50)      11300

 layer_60 (Conv2DTranspose)  (None, 128, 128, 75)      33825

 layer_70 (Conv2D)           (None, 128, 128, 3)       2028

=================================================================
Total params: 52803 (206.26 KB)
Trainable params: 52803 (206.26 KB)
Non-trainable params: 0 (0.00 Byte)
```

A Decoder architektúrát ennél jobban már nem kívánom kifejteni, hiszen a legtöbb aspektust már leírtam az Encoder esetén. Amit fontos még tudni itt, ahogy azt már korábban is említettem, a Decoder a bemeneti rétegében az Encoder végeredményét kapja meg, így az Encoder utolsó rétegének alakja és dimenziói közvetetten meghatározzák a Decoder bemenő rétegének alakját is. Mivel az Encoder 3 db Conv2D - ezek a kétdimenziós konvolúciós háló komponensei - réteget tartalmazott, így azon rétegek dekonvolúciós megfelelője a Conv2DTranspose, amely fordított irányba növeli a látens tér dimenzióit, tehát esetünkben egy skálázás történik. A végén azonban szükség volt ezt a kimenetet átalakítani egy könnyen feldolgozható formába, azaz vissza kellett kapni a bemenő kép csatornáit és méretét. Ezt egy egyszerű Conv2D réteg megoldotta az utolsó rétegben azzal, hogy a látens tér dimenzióit 75-ről egyből 3-ra csökkentem, a képméreteket pedig meghagytam az eredeti értékén. A végeredmény egy egyszerű 2D képponthalmaz, amelyet a korábban említett fejezetekben egy egyszerű színalapú bináris küszöbérték algoritmussal normalizáltam és ezt az eredményt a TensorFlow beépített DBSCAN klaszterező algoritmusának továbbadva megkaptam a lehetséges objektumok számát a bemenő képen.

3.6.3. A tanuló algoritmus

Ahhoz, hogy egy képszegmentációs modellt elkészítsünk, önmagában nem elég az MI modell. Összegezni szeretném itt, hogy eddig milyen nagyobb komponenseket érintettünk és azok milyen fontos jellemzőkkel bírnak.

A modell tanításához szükség van rengeteg tanulóadatra, jelen esetben képekre. Legalább akkora mennyiségű, amelyből létrehozható 3 halmaz. Az első halmaz magához a tanításhoz kell, a második a teszteléshez, melyet a TensorFlow implementációja használ fel belül az ellenőrzéshez, és egy validációs halmaz, amely arra kell, hogy mi magunk is ellenőrizni tudjuk a végeredményt és állítani tudjunk a kimenet alapján a hiperparamétereken. Ezeket az adatokat egy adatgenerátor komponens állítja elő, ezt mutattam be először.
Ezek után szükség van egy modell architektúrára, amelyet az előző fejezetben fejtettem ki, és szükség van egy olyan kódra, amely a definiált architektúra segítségével a generált adatokat megtanítja egy mesterséges modellnek úgy, hogy az a lehető legkevesebb veszteséggel tudja elvégezni a feladatát, jelen esetben a képszegmentálást. A veszteség egy hibafüggvény alapján számolódik és úgy is felfogható, mint a tanulás során jelenlévő, folyamatosan frissülő iránytű, amely az éppen tanuló mesterséges modellnek olyan fontos jellemzőket mutat meg, hogy jelenleg mennyire optimális maga a modell és milyen irányba kell továbbtanulnia ahhoz, hogy az iránytű a megfelelő irányba mutasson. Az optimális állapot elérése nehéz, egy mesterséges intelligenciát használó problémánál sosem lehet 100%-os, azaz nem lehet teljesen kizárni a tévedés, vagy más néven az inkorrekt MI hallucináció esetek megjelenésének esélyét.
Ha mindez rendelkezésünkre áll, és megtanítottunk egy modellt, akkor a hozzá tartozó súlyértékeket el tudjuk menteni. Ez azért hasznos, mert így nem kell minden alaklommal újratanítani az MI modellt, hiszen elegendő betölteni a már megtanított súlyhalmazt. A modell ezután nagyon gyorsan képes a döntéshozatalra, hiszen a továbbiakban már nem tanítjuk, tehát nem frissítjük a súlyokat, ha egy bemenet esetén a vártnál rosszabb végeredménnyel tér viszza a hívó félhez.
Amire még fontos odafigyelni egy Autoencoder esetén, hogy itt tulajdonképpen nem egy darab modellt kell megtanítanunk, hanem egy kódolásra képes és egy dekodolásra képes duális párt is létre kell hoznunk, hiszen maga az architektúra is erre a két fő részre osztható fel.

```
def train_model():
  autoencoder = Autoencoder(CONSTANTS["COUNT_OF_LATENT_DIMENSIONS"])
  autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

  if is_file_exists(CONSTANTS["CHECKPOINT_PATH_ENCODER"]) and
is_file_exists(CONSTANTS["CHECKPOINT_PATH_DECODER"]):
    autoencoder.encoder.build(input_shape=CONSTANTS["INPUT_SHAPE"])
    autoencoder.decoder.build(input_shape=(1, CONSTANTS["IMAGE_WIDTH"],
CONSTANTS["IMAGE_HEIGHT"], 25))
    autoencoder.encoder.load_weights(CONSTANTS["CHECKPOINT_PATH_ENCODER"])
    autoencoder.decoder.load_weights(CONSTANTS["CHECKPOINT_PATH_DECODER"])
  else:
    train_file_paths =
get_file_paths_by_directory(CONSTANTS["TRAIN_DIRECTORY"])

    train_dataset = tf.data.TFRecordDataset(train_file_paths,
compression_type='GZIP')
    train_dataset = train_dataset.map(decode_image_from_tfrecord)

    test_file_paths = get_file_paths_by_directory(CONSTANTS["TEST_DIRECTORY"])
    test_dataset = tf.data.TFRecordDataset(train_file_paths,
compression_type='GZIP')
    test_dataset = test_dataset.map(decode_image_from_tfrecord)

    autoencoder.fit(train_dataset, epochs=CONSTANTS["NUMBER_OF_EPOCHS"],
batch_size=CONSTANTS["BATCH_SIZE"], shuffle=False,
validation_data=test_dataset)
    autoencoder.encoder.save_weights(CONSTANTS["CHECKPOINT_PATH_ENCODER"])
    autoencoder.decoder.save_weights(CONSTANTS["CHECKPOINT_PATH_DECODER"])

  return autoencoder
```

A fenti kód azt a metódust reprezentálja, amelyet a modell tanítására használtam. Miután létrehozok egy objektumpéldányt az általam felülírt Autoencoder osztályból, meghívom a compile metódust, amely lefordítja az architektúrát, így az értelmezhetővé válik a TensorFlow számára is. A compile() metódus paramétereként lehet megadni az optimalizálófüggvényt is, amely arra szolgál, hogy a tanulás során a paramétereket frissítse. Én az “adam” optimalizálót választottam, mert ismereteim alapján ez volt a legelterjedtebben használt optimalizáló, és az én prototípusomnál ez az optimalizálás remekül működött. Itt lehet megadni még a hibafüggvényt is, amely esetén szintén a legpopulárisabb módszerre esett a választásom, ez pedig az MSE hibafüggvény, amely a következőképpen néz ki:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE = mean squared error
n = number of data points
Y_i = observed values
$\hat{Y}_i$ = predicted values

Ezután szükségem volt egy map metódusra, ezt egy korábbi fejezetben bemutattam, de lényegében ez azért kellett, hogy a tfrecord.gz fájlokat a tanulás során az algoritmus kicsomagolja és egyenként használja fel, hiszen ha ezt nem tudná megtenni, akkor minden adatot egyszerre kellene betölteni a memóriába, amely túlzó mértékű erőforrásigénnyel rendelkezne.
A következő metódus a modell fit függvénye, amely magát a tanulást végzi el a megadott epoch-ok számának és batchméret függvényében. A legkisebb tanulási hibát 6 epoch-al és 10-es batchmérettel értem el 30 tanulóadat és 10 tesztadat felhasználása esetén. Összekeverni az adatokat már nem kellett, hiszen a generálás során ez közvetlenül megtörtént, ezért a shuffle paraméter értékét false-ra állítottam.
A végén elmentettem az Encoder és a Decoder súlyait is egy-egy .h5 kiterjesztésű fájlba, melyet a TensorFlow a későbbiekben be tudott tölteni, így nem kell minden alkalommal újratanítani a modellt. Az elmentett modell meglétének ellenőrzése a saját magam által implementált is_file_exists metódussal történik meg, és ha létezik mind a két modellfájl, akkor a load_weight metódussal azokat be tudtam tölteni az Autoencoder objektumpéldány számára. Ahhoz azonban, hogy a betöltés működjön szükség volt tudatni a modellel azt is, hogy milyen alakú adatokkal kell dolgoznia, hiszen minden implementációm dinamikus dimenziókon alapszik, annak érdekében, hogy a prototípus később könnyebben bővíthető legyen.

4. KONCEPCIÓ MEGÍTÉLÉSE

Ahhoz, hogy meg tudjam ítélni a koncepció működésének jellemzőit, egy másik, egyszerűbb implementációval hasonlítom össze az Autoencoder erejét. Ez a módszer egy egyszerű template matching algoritmus, amelyet korábban már bemutattam, ezért csak egy gyors ismétlést említenék meg itt.

A template matching algoritmus segítségével egy minta előfordulásait meg lehet vizsgálni egy képen. Implementálni több módon is lehet, én a TensorFlow beépített megvalósítását használtam fel. A működése röviden annyiból áll, hogy a mintaként szolgáló entitást rá kell helyezni a kép egy adott pontjára és ki kell számolni egy súlyértéket, amely megmondja, hogy mennyire illeszkedik a minta az adott pozícióban. A mintát érdemes minden képpontra (sorra és oszlopra) kiszámolni és ezzel kapunk egy súlyhalmazt, amelyből ki lehet választani egy határérték szerint azt, hogy melyeket fogadjuk el és melyeket vetjük el.

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

image =
cv2.imread('/content/gdrive/MyDrive/School/resources/raw_images/independent_ba
ckup/98caea16-f75b-459c-b7e1-3d00de57de41-input.png', cv2.IMREAD_COLOR )
template =
cv2.imread('/content/gdrive/MyDrive/School/resources/target_objects/Sunflower-
09.png', cv2.IMREAD_COLOR)
template = cv2.resize(template, (16, 16))

h, w = template.shape[:2]
method = cv2.TM_CCOEFF_NORMED
threshold = 0.7
max_val = 1

while max_val > threshold:
  res = cv2.matchTemplate(image, template, method)
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

  image[max_loc[1]:max_loc[1]+h, max_loc[0]:max_loc[0]+w, :] = [0, 0, 255]

cv2_imshow(image)
```

A fenti kód egy egyszerű implementációja a Template-Matching algoritmusnak, amely egy képen illesztési módszerrel a megadott minta alapján megkeresi a képen lévő objektumokat.

Az implementációban az OpenCV által biztosított matchTemplate metódust használtam. Paraméterként meg kell adni a háttérképet, a mintaképet, amely nem lehet nagyobb méretű, mint maga a háttérkép és az illesztési metódust. Ha a minta nagyobb méretű, mint maga a háttérkép, akkor a kód hiba nélkül lefut, azonban nem fog találni objektumot a háttérképen. A method paraméter az illesztéshez használatos függvény típusát reprezentálja. Ezek a következők lehetnek:

- TM_SQDIFF
- TM_SQDIFF_NORMED
- TM_CCORR
- TM_CCORR_NORMED
- TM_CCOEFF
- TM_CCOEFF_NORMED

Habár kipróbáltam több függvényt is, a legjobb választásnak a “TM_CCOEFF_NORMED” bizonyult. A legtöbb függvény ugyanis egyáltalán nem azokat az objektumokat találta meg a képen, amelyek a valódi entitást reprezentálják.

TM_CCOEFF_NORMED Python: cv.TM_CCOEFF_NORMED	$$R(x,y)=\frac{\sum_{x',y'}(T'(x',y')\cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'}T'(x',y')^2\cdot\sum_{x',y'}I'(x+x',y+y')^2}}$$

TM_CCOEFF_NORMED függvény az OpenCV dokumentációjából

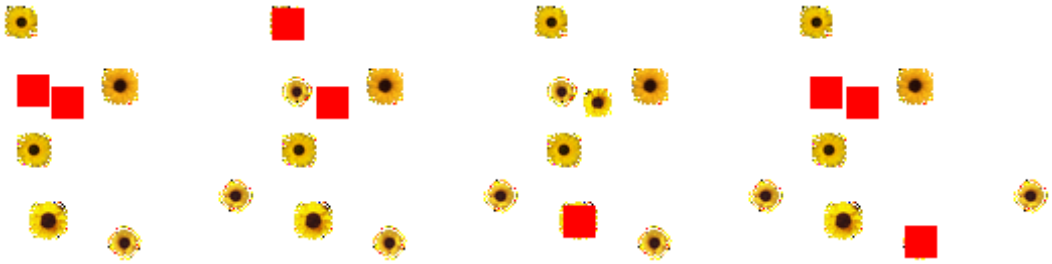
A “TM_CORR” és a “TM_SQDIFF” pedig úgy tűnt, hogy végtelen futású ciklusba kerültek.

TM_CCORR Python: cv.TM_CCORR	$$R(x,y)=\sum_{x',y'}(T(x',y')\cdot I(x+x',y+y'))$$
with mask:	$$R(x,y)=\sum_{x',y'}(T(x',y')\cdot I(x+x',y+y')\cdot M(x',y')^2)$$
TM_SQDIFF Python: cv.TM_SQDIFF	$$R(x,y)=\sum_{x',y'}(T(x',y')-I(x+x',y+y'))^2$$
with mask:	$$R(x,y)=\sum_{x',y'}((T(x',y')-I(x+x',y+y'))\cdot M(x',y'))^2$$

TM_CCORR és TM_SQDIFF függvények az OpenCV dokumentációjából

A “threshold” változót - amely azt a küszöbértéket reprezentálja, amely felett elfogadható egy objektum illesztési súlya - tapasztalati tényező alapján állítottam be 0.7 értékre. Alacsonyabb

vagy magasabb érték esetén az illesztés nem találta meg a megfelelő számú objektumot vagy túl sok helytelen illesztési eredmény volt.
A kód végén látható, hogy az OpenCV minMaxLoc függvényét használtam a megtalált entitások dimenzióinak lekérdezésére. Ebből nekem csak az objektum maximális lokális értékére volt szükségem, amelyből ki tudtam számolni a minta kép méreteit ismerve, hogy hol található a befoglaló doboz. A végén egy cv2_imshow függvénnyel megjelenítettem az eredményt.



Természetesen az én implementációm itt is csak egy prototípus erejéig nyúl, de jól látható, hogy a Template-Matching implementációja jóval egyszerűbb is lehet egy teljes modell betanításától. A probléma ezzel az, hogy amíg az Autoencoder meg tudja tanulni az objektum rejtett jellemzőit, addig a Template-Matching erre nem képes, hiszen minden objektum felismeréséhez szükség van egy reprezentáló mintára, amely a természetből nehezen vagy egyáltalán nem kinyerhető.
Egy másik fontos aspektus még az, hogy a Template-Matching algoritmus minden mintát külön-külön próbál illeszteni a képre. Ez azért okozhat problémát, mert a futási ideje szignifikánsan megnőhet abban az esetben, ha sok minta van és a kép mérete nagyobb, mint amit a prototípusban használtam.

Lehetőségek a jövőben

Habár a dolgozat által leírt munkám jelenleg nem számottevő a nagyvilágban, én úgy gondolom minden projektnek van értelme, ha abból tanulni is lehet. Nem tudom még mit hoz számomra a jövő, de ha lehetőségem nyílik kiterjeszteni a képességeit az általam létrehozott modellnek, akkor a jelenleg mindennapi technológiákat is felhasználva úgy gondolom, hogy képes leszek fontos globális mutatókat is fellelni a világban, amelyek számottevő felismeréshez vezethetnek.

Zárszó

Napjainkban Földünk globális gazdasági statisztikai mutatóinak előrejelzése fontos döntések meghozatalában segíthet. A globalizáció megjelenésének köszönhetően egyre követhetőbbé válik, hogy milyen folyamatokkal kell szembenéznünk és azok milyen mértékben befolyásolják a Földön élő populációk életkörülményeit. Sajnos azonban még mindig nem rendelkezünk elég ismerettel ahhoz, hogy reális jóslatokkal felkészüljünk a jövőben látni vélt globális katasztrófák és hatalmas gazdasági változások által generált akadályokkal szemben. Szerencsére azt a korszakot tapasztalhatjuk magunk körül, ahol a mesterséges objektumok önmagukban testesítik meg az általunk ismert összes tudományág egyesülését, ezért fontos, hogy képesek legyünk kontrollálni azokat, mert a végtelenül komplex rendszerek karbantartása és továbbfejlesztése egyre nehezebbé válik. Feltehető a kérdés tehát, hová tart a világ és milyen eszközeink lesznek pár évtized múlva?

Irodalomjegyzék

- OpenCV: Object Detection
  - https://docs.opencv.org/4.x/df/dfb/group__imgproc__object.html#gga3a7850640f1fe1f58fe91a2d7583695dac6677e2af5e0fae82cc5339bfaef5038
  - Utolsó megtekintés dátuma: 2023.12.09.
- Intro to Autoencoders | TensorFlow Core
  - https://www.tensorflow.org/tutorials/generative/autoencoder
  - Utolsó megtekintés dátuma: 2023.12.09.
- Képfeldolgozás haladóknakpdfauthor=Palágyi Kálmán (oszk.hu)
  - https://oszkdk.oszk.hu/storage/00/01/42/15/dd/1/Palagyi_Kepfeldolgozas.pdf
  - Utolsó megtekintés dátuma: 2023.12.09.
- 3. Geometriai transzformációk (u-szeged.hu)
  - http://www.inf.u-szeged.hu/~kato/teaching/DigitalisKepfeldolgozasTG/10-ShapeDetection.pdf
  - Utolsó megtekintés dátuma: 2023.12.09.
- Slide 1 (u-szeged.hu)
  - https://www.inf.u-szeged.hu/~tothl/ann/07.%20Konvolucios%20neuronhalok.pdf
  - Utolsó megtekintés dátuma: 2023.12.09.
- Make Your Own Neural Network: Calculating the Output Size of Convolutions and Transpose Convolutions
  - http://makeyourownneuralnetwork.blogspot.com/2020/02/calculating-output-size-of-convolutions.html
  - Utolsó megtekintés dátuma: 2023.12.09.
- machine learning - Why COST FUNCTION AND MSE IS CALLED THE SAME? - Data Science Stack Exchange
  - https://datascience.stackexchange.com/questions/122066/why-cost-function-and-mse-is-called-the-same
  - Utolsó megtekintés dátuma: 2023.12.09.
- Implementing a View - Xamarin | Microsoft Learn
  - https://learn.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/view#creating-the-custom-renderer-on-android
  - Utolsó megtekintés dátuma: 2023.12.09.
- A detailed example of data generators with Keras (stanford.edu)
  - https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly
  - Utolsó megtekintés dátuma: 2023.12.09.
- tf.data.Dataset | TensorFlow v2.14.0
  - https://www.tensorflow.org/api_docs/python/tf/data/Dataset
  - Utolsó megtekintés dátuma: 2023.12.09.
- Free Sunflower Clipart Transparent Background, Download Free Sunflower Clipart Transparent Background png images, Free ClipArts on Clipart Library (clipart-library.com)
  - https://clipart-library.com/free/sunflower-clipart-transparent-background.html#google_vignette
  - Utolsó megtekintés dátuma: 2023.12.09.

- New technique based on 18th-century mathematics shows simpler AI models don't need deep learning (techxplore.com)
    - https://techxplore.com/news/2023-10-technique-based-18th-century-mathematics-simpler.amp?fbclid=IwAR0pgB32Gpvr9HXlbrWFw0Y-6locGoEAW1IYOk-SneM4O10owmNOaKKOPwY
    - Utolsó megtekintés dátuma: 2023.12.09.
- Template Matching — Image Processing | by Matt Maulion | Medium
    - https://mattmaulion.medium.com/template-matching-image-processing-6eb1d5425248
    - Utolsó megtekintés dátuma: 2023.12.09.
- OpenCV ApproxPolyDP | Learn the Examples of OpenCV ApproxPolyDP (educba.com)
    - https://www.educba.com/opencv-approxpolydp/
    - Utolsó megtekintés dátuma: 2023.12.09.
- R-CNN | Region Based CNNs - GeeksforGeeks
    - Utolsó megtekintés dátuma: 2023.12.09.
- OpenCV: Contours : Getting Started
    - https://www.geeksforgeeks.org/r-cnn-region-based-cnns/
    - Utolsó megtekintés dátuma: 2023.12.09.
- A Complete Understanding of Dense Layers in Neural Networks (analyticsindiamag.com)
    - https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/#:~:text=one%20by%20one.-,What%20is%20a%20Dense%20Layer%3F,in%20artificial%20neural%20network%20networks.
    - Utolsó megtekintés dátuma: 2023.12.09.
- Képfeldolgozás haladóknakpdfauthor=Palágyi Kálmán (oszk.hu)
    - https://oszkdk.oszk.hu/storage/00/01/42/15/dd/1/Palagyi_Kepfeldolgozas.pdf
    - Utolsó megtekintés dátuma: 2023.12.09.
- autoencoder.ipynb - Colaboratory (google.com)
    - https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/autoencoder.ipynb#scrollTo=iYn4MdZnKCey
    - Utolsó megtekintés dátuma: 2023.12.09.
- Install WSL | Microsoft Learn
    - https://learn.microsoft.com/en-gb/windows/wsl/install
    - Utolsó megtekintés dátuma: 2023.12.09.
- 10 Clustering Algorithms With Python - MachineLearningMastery.com
    - https://machinelearningmastery.com/clustering-algorithms-with-python/#:~:text=Cluster%20analysis%2C%20or%20clustering%2C%20is,or%20clusters%20in%20feature%20space.
    - Utolsó megtekintés dátuma: 2023.12.09.
- Building Autoencoders in Keras
    - https://blog.keras.io/building-autoencoders-in-keras.html
    - Utolsó megtekintés dátuma: 2023.12.09.
- An Achromatic Approach to Compressing CNN Filters Using Pattern-specific Receptive Fields – Nextjournal
    - https://nextjournal.com/tangcc35/model-pruning-applying-image-clustering-methods-and-weights-combinations-to-convolutional-neural-network-layers
    - Utolsó megtekintés dátuma: 2023.12.09.
- ConvNet Shape Calculator (madebyollin.github.io)

    - https://madebyollin.github.io/convnet-calculator/
    - Utolsó megtekintés dátuma: 2023.12.09.
- Unsupervised clustering with DBSCAN | by Manyi | Medium
    - https://medium.com/@manyi.yim/unsupervised-clustering-with-dbscan-17d03c439c26
    - Utolsó megtekintés dátuma: 2023.12.09.
- 3. Geometriai transzformációk (u-szeged.hu)
    - http://www.inf.u-szeged.hu/~kato/teaching/DigitalisKepfeldolgozasTG/10-ShapeDetection.pdf
    - Utolsó megtekintés dátuma: 2023.12.09.
- Variational Autoencoders are Beautiful | Blogs (compthree.com)
    - https://www.compthree.com/blog/autoencoder/
    - Utolsó megtekintés dátuma: 2023.12.09.
- Mean Shift (ml-explained.com)
    - https://ml-explained.com/blog/mean-shift-explained
    - Utolsó megtekintés dátuma: 2023.12.09.
- meanShiftR | R-bloggers
    - https://www.r-bloggers.com/2016/08/meanshiftr/
    - Utolsó megtekintés dátuma: 2023.12.09.
- K-means clustering with Amazon SageMaker | AWS Machine Learning Blog
    - https://aws.amazon.com/blogs/machine-learning/k-means-clustering-with-amazon-sagemaker/
    - Utolsó megtekintés dátuma: 2023.12.09.

Nyilatkozat

Alulírott Pörzsölt Krisztián, programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, Programtervező Informatikus MsC diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Dátum

Budapest, 2023.12.12.

Aláírás

Köszönetnyilvánítás

Köszönettel tartozom Dr. Bodnár Péter témavezetőnek amiért kitartó munkával és sok hasznos meglátással támogatta a munkámat. Köszönöm továbbá szerető szüleimnek, édesanyámnak és édesapámnak, akik nélkül nem juthattam volna el idáig és remélem további sikereimet is jó szívvel tudják majd fogadni. Köszönöm Rubóczki Rékának a számomra igen érdekes problémakör felvetését, amelyből nem születhetett volna meg ezen dolgozat alapötlete.