FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Képfeldolgozási algoritmusok alkalmazása és vizsgálata

Neurális hálókkal

Szakdolgozat

Készítette
Pongrácz Vince Balázs

Konzulens
dr. Ekler Péter

2022. december 8.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Pongrácz Vince Balázs*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. december 8.

Pongrácz Vince Balázs
hallgató

Kivonat

Napjainkban szinte minden a mesterséges intelligenciáról és a mély tanulásról szól. Elég a Google és facebook féle ajánlórendszerekre, az Alexa, Google vagy Siri asszisztensek beszédfelismerésére vagy az arcfelismerőkre gondolni. Ahol lehetséges, megpróbálják kihasználni az óriási adatmennyiség és a gépi tanulás erősségeit.

Szakdolgozatomban egy retusálást segítő alkalmazás létrehozása a cél, ami fotósokat segít bizonyos képtulajdonságok prediktálásával. A probléma megoldásához pedig konvolúciós neurális hálókat (convolutional neural networks, CNN) használok.

A profi fotózásban a legtöbb esetben nyersképek készülnek, ez egy kamera és gyártóspecifikus formátum, ami a szenzorról kiolvasott nyers információt tárolja. Ez előnyös, mert így a képjellemzők széles skáláján lehet utólag változtatni, hiszen a feldolgozatlan adat áll rendelkezésre, míg a legtöbb képkódolási formátum vagy veszteséges, és/vagy bizonyos tulajdonságait (például fehéregyensúly) a képnek beégeti, ezáltal nem módosítható. A legtöbb esetben a fehéregyensúly (white balance) és a lila-zöld (tint) azok a tulajdonságok, amiket a kamera nem feltétlenül tud helyesen beállítani, e mögött a fénymérés tökéletlensége áll. Ezen kívül a kép fényességének (exposure), kontrasztjának (contrast), illetve vibrálásának (vibrance – ez a szaturáció egy enyhébb formája) prediktálása is cél, tisztán kényelmi szempontokból.

Dolgozatomban a képek előzetes feldolgozásáról, a képfeldolgozásban felhasznált neurális hálókról, és ezek építőelemeiről adok áttekintést. Ismertetem a hálók létrehozásához szükséges főbb könyvtárakat, mint a TensorFlow és Keras. Ezután bemutatásra kerül a kész szoftver, amely a nyersképek retusálását gyorsítja azáltal, hogy a korábban felsorolt képtulajdonságokat a nyersképek alapján megtanulja, majd ezt a tudást még ismeretlen, de hasonló képekre alkalmazza. A szakdolgozatot a megoldás során kipróbált és betanított neurális hálók elemzése zárja.

Abstract

Almost everything these days is about artificial intelligence (AI) and deep learning. It is enough to think of the recommendation systems of Google and Facebook, the speech recognition in assistants like Alexa, Google and Siri or facial recognition systems. When it is possible, it is tried to leverage the strengths of huge amount of data and machine learning.

In my thesis, the goal was to create an application that helps image retouching, which helps photographers by predicting certain image properties. I use convolutional neural networks (CNN) to solve this problem.

In most cases in professional photography, raw images are taken, this is a camera and manufacturer-specific format that stores the information read from the sensor. The advantage of this, that a broad scale of the image characteristics could be changed afterwards, since the unprocessed data is available, while most image codecs either lose and/or burn certain properties (e.g. white balance) into the image, so it cannot be modified. In most cases, white balance and tint are such properties that the camera could not set correctly, the cause behind this is the imperfection of the light metering. In addition, the goal is to predict brightness (exposure), contrast, and vibrance of the image (this is a milder form of saturation), purely for convenience.

In my thesis, I give an overview about image pre-processing, about the neural networks used in image processing, and their building blocks. I describe the main libraries needed to create neural networks, such as TensorFlow and Keras. Afterwards the finished software is presented, which speeds up the retouching of raw images in order to learn the previously listed image properties based on the raw images, and then apply this knowledge to still unknown but similar images. The thesis ends with the analysis of during the solution tested and trained neural networks.

1. fejezet

Bevezetés

A következő fejezetben röviden bemutatásra kerül a választott téma, a választás szempontjai, a személyes motiváció a téma iránt, illetve a szakdolgozat felépítése.

1.1. Témaválasztás, személyes motiváció

Az utóbbi évtizedben óriási fejlődés figyelhető meg a mesterséges intelligencia (artificial intelligence) és az adattudomány (data science) területén. Évről évre fejlesztik és teszik elérhetővé a jobbnál jobb keretrendszereket és architektúrákat, amik segítségével a programozónak már nem feltétlenül a gépi tanulás alapjainak implementálásával kell töltenie ideje nagy részét, koncentrálhat helyette a magasabb szintű, komplexebb problémák megoldására.

Az utóbbi egy években az élet számos területén figyelhető meg a mesterséges intelligencia és gépi tanulás térhódítása, illetve támogató jelenléte. Gondolhatunk itt az autókban az olyan vezetést segítő rendszerekre, mint a sávtartás és közlekedési táblák felismerése, vagy akár a nagy közösségi oldalak, videómegosztók ajánlórendszerei, amik eszméletlen pontossággal képesek megtanulni a felhasználók szokásait és pontosan azt a tartalmat a felhasználó elé tenni, amitől sok időt tölt az adott platformon, akár függőségbe hajtva és elpazarolva idejét.

Az 1.1 táblázat a Google Scholar-on Engineering & Computer Science témában fellelhető legnépszerűbb publikációk listájának első 10 elemét mutatja. Ezek között cím alapján legalább 5 olyan található, ami a mesterséges intelligencia, gépi tanulás, vagy mély tanulás témájába esik, vagyis a mesterséges intelligencia jelenleg is nagyon gyorsan és dinamikusan fejlődő területe az informatikának, ezért mindenképp érdemes vele foglalkozni.

	Publication	h5-index	h5-median
1.	IEEE/CVF Conference on Computer Vision and Pattern Recognition	389	627
2.	Advanced Materials	312	418
3.	International Conference on Learning Representations	286	533
4.	Neural Information Processing Systems	278	436
5.	IEEE/CVF International Conference on Computer Vision	239	415
6.	International Conference on Machine Learning	237	421
7.	Renewable and Sustainable Energy Reviews	227	324
8.	Advanced Energy Materials	220	300
9.	ACS Nano	211	277
10.	Journal of Cleaner Production	211	273

1.1. ábra. A Google Scholar Engineering & Computer Science kategória legnépszerűbb publikációi [40]

Ezen kívül lenyűgöztek korábbi tanulmányaim alatt a mesterséges intelligencia képességei, hogy mennyire pontosak, valósak, enyhe túlzással emberiek tudnak lenni, ezért kezdtem el mélyebben ezzel a területtel, azon belül is a mély tanulással foglalkozni.

A képfeldolgozás területéről választottam a témát, egy retusálást segítő alkalmazást tűztem ki célul. Nagy szenvedélyem a fotózás, évek óta űzöm ezt a hobbit, tagja vagyok a SPOT fotókörnek. Aki kevésbé jártas a fotózásban, azt hiheti, hogy amint lenyomták az exponálógombot és kattant a gép, kész is a kép, a fotós munkája csak pár ezredmásodpercig tartott. Ez a valóságban nem így van, ezután jön a fotózás hosszabb része, a képek retusálása. Ez a folyamat általában jóval több időt vesz igénybe, mint maga a fotózás, néha igen pepecselős munka. Hiába vannak jól bevált presetek (előbeállítások), amik segítségével hasonló képeket egységesen be lehet állítani, ez a megoldás az összes képre ugyan azokat a beállításokat alkalmazza, egy sorozatban viszont nagyon különböző képek, nagyon különböző beállításokkal fordulhatnak elő.

Az ötlet az utómunkázás emberi részének részleges automatizálása, a gyakrabban változó képparaméterek minden képre egyénileg való prediktálásával. Ilyen, gyakran képenként változó paraméterek például a fehéregyensúly (white balance, WB), az árnyalat (tint) (ez a lila-zöld egyensúly beállítása), a világosság (exposure), a kontraszt (contrast), illetve a vibrálás (vibrance). A szakdolgozat keretében a retusálás emberi részének gyorsítása és automatizálása volt cél, ehhez célszerű választás volt gépi tanulás és neurális hálók alkalmazása. Alkalmazásom a fent említett képtulajdonságokat prediktálja a nyersképekből kiindulva, majd ezek az értékek képenként korrekciós fájlokba kerülnek. Az Adobe Lightroom ezeket a korrekciós fájlokat importálásánál a nyersképekkel együtt feldolgozza és minden képre a saját korrekcióját alkalmazza, ezáltal egy előzetes, képre szabott beállítást adva. A hatás olyan, mintha minden képre egyéni preset kerülne, végeredményben pedig a retusálás egy része automatizált.

1.2. A szakdolgozat felépítése

A szakdolgozat következő fejezetében a mesterséges intelligencia, gépi tanulás, neurális hálók és mély tanulás rövid bevezetője következik. A 3. fejezetben az alkalmazás és a modellek követelményei találhatók. A 4. fejezet a felhasznált technológiákat és eszközöket részletezi. Az 5. fejezetben az alkalmazás fejlesztése áll középpontban, a 6. fejezet a

modellek leírását, elemzését és a modellválasztást tartalmazza, míg a 7. fejezet a projekt értékelését, továbbfejlesztési lehetőségeit és ötleteit tartalmazza.

2. fejezet

Mesterséges intelligencia, gépi tanulás, neurális hálók és mély tanulás

Az embereket mindig is foglalkoztatta, hogy hogyan lehet a gépeket emberhez hasonló intelligenciával ellátni, így a gépi tanulás ötlete sem ma fogalmazódott meg először, de napjainkban óriási lendületet kapott a temérdek adatnak és a megnövekedett számítási kapacitásnak köszönhetően. Ugyanakkor a matematikai, elméleti alapokat már az 1950-es évektől kezdve elkezdték lerakni.

1956 nyarát tekintik a mesterséges intelligencia, mint tudományos terület születésének. Ekkor volt az Amerikai Egyesült Államokban, Dartmouthban egy konferencia (Dartmouth Summer Research Project on Artificial Intelligence), ahol többek között olyan tudósok, kutatók, mint Marvin Minsky (később az MIT AI laboratory egyik alapítója, neurális hálókat kutatta, Perceptrons című könyve sokáig meghatározó alapmű volt a neurális hálók analízisében) [28], John McCarthy (Lisp programozási nyelv tervezője, a Lisp volt a korai mesterséges intelligenciát alkalmazó alkalmazásokhoz használt nyelv, garbage collection kitalálója, Stanford AI laboratory egyik alapítója) [24], vagy Claude Shannon (információelmélet atyja, az mesterséges intelligencia területén a Shannon's Mouse-t érdemes említeni: egy mechanikus robotegér, ami megtalálja a kiutat a labirintusból) [11] találkoztak és gondolkodtak arról, hogy mik egy mesterséges intelligencia követelményei. Többen közülük később maradandót alkottak ebben az akkor frissen született tudományágban.

A mesterséges intelligencia egyik definíciója szerint a mesterséges intelligencia az olyan számítógépes rendszerek fejlesztését és elméletét jelenti, amik képesek megoldani feladatokat, amik normál esetben emberi intelligenciát igényelnek. Ilyen feladat például a beszédfelismerés, a vizuális észlelés, döntéshozás és a nyelvek közötti fordítás. [6]

2.1. Gépi tanulás

A mesterséges intelligencia egyik ága a gépi tanulás. Egy rendszer gépi tanulást valósít meg, ha képes a tanulásra, azaz felismeri az adott adatban lévő mintázatokat, majd ezek alapján a még ismeretlen, de hasonló adatokra vonatkozó következtetéseket képes levonni. Ezen kívül a tanulás alatt képes visszajelzések alapján változtatni és javítani saját viselkedésén és kimenetein.

A gépi tanulást általában 3 területre osztják fel, ezek a supervised learning (felügyelt tanulás), unsupervised learning (nem felügyelt tanulás), és a reinforcement learning (megerősítéses tanulás).

A supervised learning esetén az adathalmaz a bemenetekből és az ezekre elvárt kimenetekből áll, a cél pedig a bemenet-kimenet párokból megtanulni egy általánosítást, ami a még látatlan, de hasonló bemenetekre is teljesül.

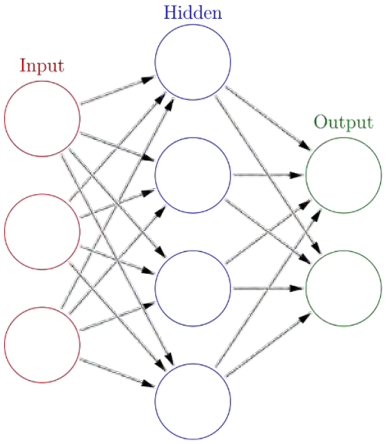
Unsupervised learning esetében a bemeneti adatokhoz nem tartozik elvárt kimenet, itt az adatokban fellelhető struktúrát, mintázatokat kell felismerni, majd ez alapján lehet szabályokat kikövetkeztetni.

A reinforcement learning egy tanuló modellből és a környezetből áll. A modell a bemenetekre vagy ingerekre akciókkal reagál, ez a kimenet. Az akciók alapján a környezet visszajelzéseket ad a modellnek, amit a modell a megkap. A modell ezután a saját belső állapotát és a visszajelzést figyelembe véve módosít viselkedésén. [18]

2.2. Neurális hálók és mély tanulás

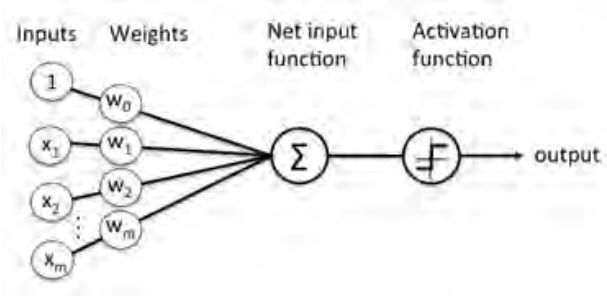
(neural networks and deep learning)

A neurális háló napjainkban az egyik legelterjedtebb gépi tanulási modell. Használattól függően megvalósíthatnak supervised és unsupervised learning-et is, a szakdolgozatban supervised learning-ről lesz szó. A neurális hálók modellje sok hasonlóságot mutat az emberi aggyal, abban is sok kisebb egység, azaz neuron kapcsolódik egymáshoz, ezáltal egy bonyolult hálót alkotva, amelyekkel komplex problémák oldhatók meg.



2.1. ábra. Egy egyszerű neurális háló sematikus rajza [7]

A neurális hálónak van egy bemeneti rétege (input layer), rejtett rétegei (hidden layers) és egy kimeneti rétege (output layer). Ezek a rétegek egyszerű feldolgozóegységekből, neuronokból állnak, mindegyik neuronnak több bemenete és kimenete van. Ezen kívül a neuronokhoz tartoznak súlyok (weights), amik a bemenetet súlyozzák, illetve egy aktivációs függvény (activation function), ami a neuron kimenetét határozza meg.



2.2. ábra. Egy neuron felépítése [33]

A háló kimenete alapján két fajta neurális hálót különböztethetünk meg. Ha a kimenet diszkrét, akkor a neurális háló osztályozási feladatra lett tervezve, osztályozásról (classification) beszélünk. Egy jó példafeladat erre, amikor el kell dönteni, hogy kutya vagy macska van egy képen. Amennyiben a kimenet folytonos, azaz (elméletileg) végtelen sok értéket vehet fel, akkor regresszióról (regression) beszélünk. Szakdolgozatomban regressziós feladatra használok neurális hálókat.

A neurális hálók tanításához rengeteg adat kell, ez a működésük alapja. Minél több olyan adat áll rendelkezésre, amihez hasonlót utána a valóságban is látni fog a háló, annál hatékonyabb lesz.

A tanítás során fontos, hogy a bemenetekhez az elvárt kimenetek is ismertek legyenek. A tanulás folyamán a hálónak bemenetként adott adatokra kapott kimeneteket hasonlítjuk össze az előre elvárt kimenetekkel, majd a két adat összehasonlításából egy hibafüggvény (error function) segítségével képzett hibaérték (loss) alapján frissítjük a háló súlyait, ezt hívják backpropagation-nek. Technikailag ez a megoldás az egyszerűbb és a gyakorlatban elterjedt. Megjegyzem, hogy az aktivációs függvények tanítása is egy járható út lenne, folynak kutatások erre vonatkozóan. Jelenleg ez nem elterjedt, de az eddigi eredmények azt mutatják, hogy az aktivációs függvények tanításával kevesebb rétegből, de hosszabb tanítási idővel érhető el ugyan az a pontosság, mint a szokásos neurális hálóknál, ahol csak a súlyokat frissítik. A súlyok frissítésében kulcsszerepe van még az optimalizálóknak (optimizer), ami a súlyfrissítés ütemét, a tanulási rátát (learning rate) befolyásolja. [5]

A hálók előre irányát használva a kimeneteket kapjuk meg. Amikor a hálót már éles környezetben alkalmazzák, mert már kielégítő pontossággal adja meg azokat az értékeket, akkor csak ezt az előre irányt használják. A tanítás fázisában viszont fontos a visszajelzés, ezért a kimenetről információt kell visszajuttatni a neuronoknak, ami alapján egy következő minta alkalmával már jobb értékeket fognak prediktálni.

A mély tanulás (deep learning) szintén a neurális hálókra épít, itt a hangsúly a nagyszámú rejtett rétegen van, azaz a hálózat mélysége miatt hívják mély tanulásnak. A több rétegű neurális hálók előnye a nem neurális háló alapú modellekkel szemben, hogy a képek jellemzői automatikusan kerülnek felismerésre, nem kell külön lépésben a jellemzők összegyűjtésével (feature extraction) foglalkozni, nem kell manuálisan megkeresni, hogy mely jellemző befolyásolja azt, amit prediktálni szeretnénk – ez automatikusan kialakul a rétegekben. Például a képfeldolgozási feladatoknál az alsóbb rétegekben az élek, vagy a sötétebb/világosabb részei a képnek kerülnek felismerésre, míg a későbbi rétegekben konkrétabb alakzatokat ismer fel a háló, ha például alakzatfelismerés a feladat.

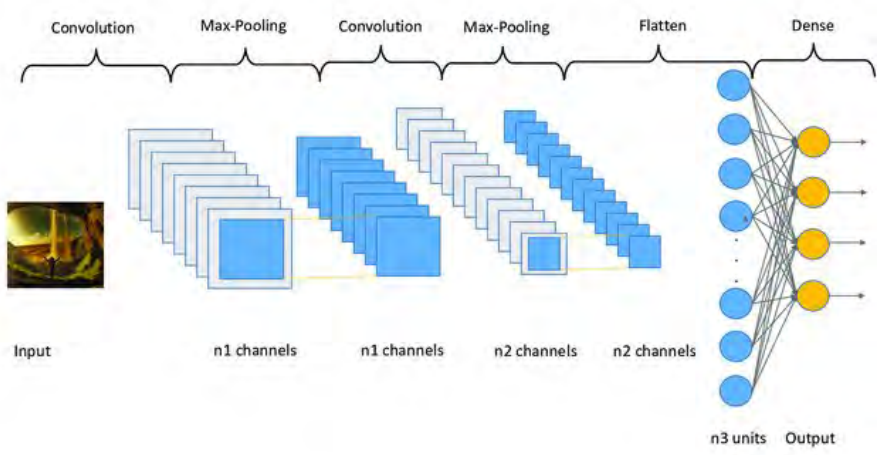
2.3. Konvolúciós neurális hálók

(convolutional neural networks, CNN)

A konvolúciós neurális hálókat főleg képfeldolgozásban használják. Működési elvükben megegyeznek a fentebb leírtakkal, a különlegesség a többdimenziós adatok kezelésében van.

Képek feldolgozása során a háló egy [x * y * z] mátrixot kap bemenetként, ahol x az egy sorban, y pedig az egy oszlopban található pixelek számát jelöli. A z koordináta az RGB színcsatorna reprezentálására szolgál, minden pixel a piros, zöld és kék színek keveréséből áll össze, így z = 3.

A konvolúciós neurális háló több konvolúciós (convolutional layer) és összevonó (pooling layer) rétegből, egy sorosító rétegből (flatten layer), majd még számos teljesen kapcsolt rétegből (fully connected layer) áll, ami a már korábban tárgyalt, mondhatni „normál" rétegeket jelenti a neurális hálóban. Az 2.4 ábrán egy általános konvolúciós neurális háló felépítése látható a különböző rétegekkel.

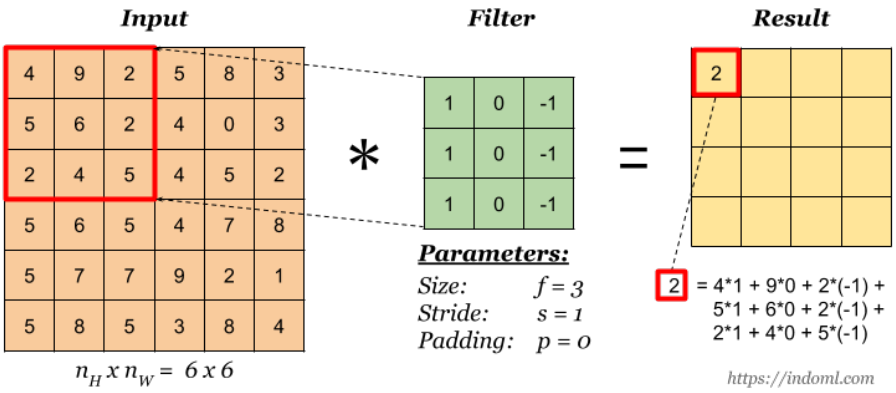


2.3. ábra. Egyszerű konvolúciós neurális háló [17]

A következő részekben a konvolúciós neurális hálók építőelemei kerülnek bemutatásra.

2.3.1. Konvolúciós réteg (convolutional layer)

A bemenet tömörítésére használják, megegyezik a jelfeldolgozásban használt konvolúcióval. Egy konvolúciós filter szintén [x*y*z] mátrixként adható meg, ahol a képhez hasonlóan x, y, z a filter dimenzióit jelölik. Ezt a mátrixot csúsztatjuk végig a képen. Az egymás felett lévő mátrixelemeket egymással össze kell szorozni, majd összeadni. Ez adja meg a következő képréteget, ezt aktivációs térképnek (activation map) nevezik, amely utána további rétegek bemenete lehet, például újabb konvolúciós réteg (convolutional layer), vagy összevonó réteg (pooling layer), végül pedig sorosító réteg (flatten layer).



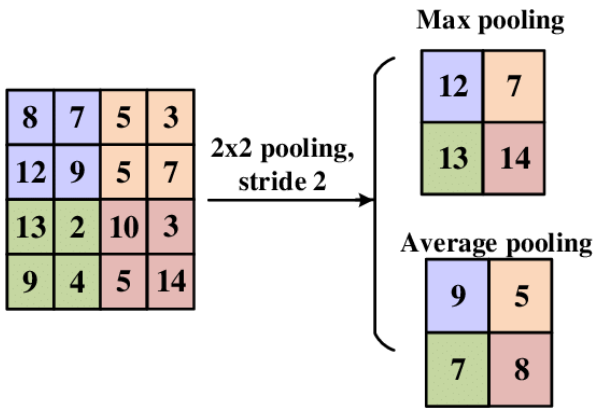
2.4. ábra. A konvolúció művelet bemutatása [21]

2.3.2. Összevonó réteg (pooling layer)

Ennek a rétegnek az aktivációs térképek dimenzióinak csökkentése a célja. Nem tanítható, csak mintavételezési feladatokat lát el. [x*y] alakú mátrixként adható meg, szintén végigpásztázza az bemenetet, majd többféle kimenetet produkálhat, a réteg fajtájától függően, a leggyakoribbak a következők:

- átlag összevonás (average pooling): Az adott [x*y] régióban lévő értékek átlagát veszi és írja ki kimenetként. Kisebb változások a bemenetben nem nagyon változtatják a réteg kimenetét.
- maximum összevonás (max pooling): Az adott [x*y] régióban lévő értékek közül veszi a legnagyobbat, éldetektálásra kifejezetten alkalmas réteg. A maximum összevonás hatékonyabbnak bizonyult a gyakorlatban osztályozási feladatok során, mert jobban kidomborított bizonyos mintázatokat a képen.

Ezen kívül még fontos paraméter a filter lépésenkénti csúsztatása, a stride. Azt adja meg, hogy hány sornyit/oszlopnyit mozdul el a filter a következő mintavételezéshez, alapértelmezetten ennek értéke egy. Amennyiben nem, úgy csúsztatott konvolúcióról beszélünk (strided convolution), ezzel pedig az összevonó rétegeket lehet helyettesíteni, hiszen több mint egyet csúsztatva ez a réteg is csökkenti a kimeneti dimenziót, mintavételez.



2.5. ábra. Példa az average és max pooling rétegek működésére [29]

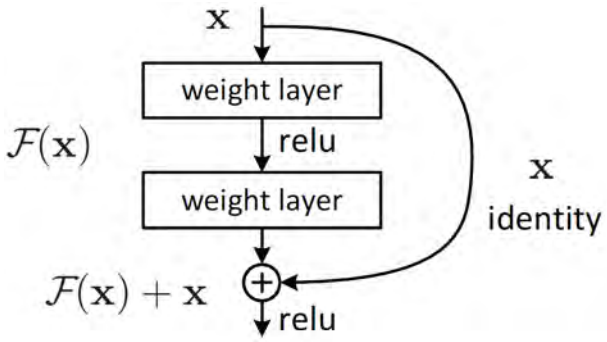
A fentebbi összevonásoknak létezik globális változója (global average pooling, global max pooling), ezt a sorosító (flatten) réteg helyett szokták alkalmazni. Ekkor bemeneten-

ként egy érték képződik. Bement ilyenkor az aktivációs térképek sokasága, minden térkép egy új értékre lesz leképezve, azaz lesz átlagolva, vagy a maximuma kiválasztva. Több aktivációs térkép esetén ez egy vektorrá áll össze. [13]

2.3.3. Visszacsatolt kapcsolatok (residual connections)

A visszacsatolt kapcsolatok (residual connection) a mély hálózatokban előjövő vanishing/exploding gradients problematikáját próbálják megoldani. A probléma gyökere, hogy a tanítási információ vagy elveszik a háló mélyebb rétegeibe érve, vagy túlságosan nagy értékeket vesz fel, és „elszáll". Egyik eset se jó, mivel a mélyebb rétegek így képtelenek a tanulásra. A probléma kiküszöbölésére olyan kapcsolatok bevezetésére van szükség, amik átugranak rétegeket, ezáltal a rövidebb visszaúton (a visszacsatolt kapcsolaton) könnyebb az információáramlás.

A mély neurális hálók képesek nagyon komplex összefüggések megtanulására, ugyanakkor a kutatási tapasztalatok azt mutatták, hogy az azonosság, mint leképezés megtanulása problémás tud lenni. A visszacsatolt kapcsolatok (residual connection) segítségével a rétegek kimenete az eredeti bemenet (vagy előző réteg által előállított aktivációs térkép [activation map]) és az átugrott rétegekben megtanult kiegészítés összegeként áll elő. Összegzés helyett a konkatenálás művelete is előfordulhat a két útvonal egyesítésére.



2.6. ábra. Residual connection[38]

Az ilyen neurális hálót hosszabb idő tanítani, mert az összeadás miatt mindig jelen van az eredeti bemenet információja is, de hasznos is, mivel az információvisszaáramlás alatt a hiba, a frissítési információ egésze eljuthat az összes réteghez, így nem veszik el tanulási információ. Az előrecsatolt (feedforward) neurális hálóknál a bemenet az összes rétegen keresztülmegy, egy útvonal van. A visszacsatolt kapcsolattal (residual connection) ellátott hálók esetében számos útvonal van, amin a bemenet a háló kimenetéhez érhet. A több útvonal, azaz a több, részben különböző háló együttes eredménye van a végén összegezve, így érthető módon nagyobb a modell komplexitása. Így igazából a vanishing/exploding gradient probléma sincs igazán megoldva, csak több sekélyebb hálóval van elfedve. A sekélyebb hálókban ugyanis nem jön elő ez a probléma, pont az alacsonyabb rétegszám miatt. [25] [41]

3. fejezet

Az alkalmazás és modellek követelményei

Az alkalmazás célja előre betanított neurális hálók segítségével még ismeretlen, de a tanítási adathalmazhoz hasonló nyersképek bizonyos tulajdonságainak helyes prediktálása, ezen tulajdonságok .xmp fájlba írása. A prediktálás után az Adobe Lightroom program megnyitása és ezáltal lehetőség még utólagos korrekciókra is cél, amennyiben a neurális hálók nem találják el jól a képtulajdonságokat.
Az alkalmazás és a neurális hálók sikerességéhez a következő követelmények teljesülése szükséges:

- **Megfelelő minőségű és mennyiségű adat:**

  A mély tanulásban nagyon fontos, hogy nagy számosságú, jó minőségű adathalmaz álljon rendelkezésünkre. Akár több millió kép is kellene tanítási halmazként egy, az esetek jelentős részét lefedő, ténylegesen hatékony és elfogadhatóan pontos végeredményhez. Sajnos ez a követelmény nem tud teljesülni, mivel a fellelhető publikus adathalmazok között nincs olyan, ahol ilyen mennyiségben tartanának nyersképeket korrekciós fájlokkal együtt, ez óriási adatmennyiség. Ennek ellenére adataugmentációval (data augmentation) megpróbálok segíteni ezen a problémán.

  A nyilvánosan elérhető projektek nagy része kisméretű (32*32-es) képek sokaságán hajt végre osztályozási feladatot. Ez az alkalmazás viszont nagyméretű képeken regressziós feladatot hajt végre, hiszen folytonos értéket prediktál.

- **Tanulás nyersképekből és a hozzájuk tartozó korrekciós fájlokból:**

  A tanulási adathalmaznak tartalmaznia kell a nyersképeket és a hozzájuk tartozó helyes .xmp fájlokat.

- **5 képtulajdonság prediktálása:**

  Ezek a képtulajdonságok a fehéregyensúly (white balance), a zöld-lila árnyalat (tint), világosság (exposure), kontraszt (contrast) és a vibrálás (vibrance).

- **Megfelelő neurális háló modell:**

  A modellnek elfogadhatóan pontos értékeket kell prediktálnia. A modell komplexitásának meghatározása bonyolult feladat, sok múlik a felhasznált kiindulási modellen, felhasznált rétegek típusán és számán, neuronok számán, tanulási rátán (learning rate), bemeneti adatokon, hibafüggvényeken (loss functions), optimalizálókon (optimizers) és tanulási ciklusok (epochs) számán. Ezek kiválasztása hosszadalmas feladat.

- **A prediktált tulajdonságok képszerkesztő által feldolgozhatók:**

  A prediktált értékekből .xmp fájlt kell generálni a nyersképpel megegyező névvel. Így amikor a Lightroom importálja a nyersképeket, rögtön alkalmazza a mellette található korrekciós fájlt is, ez egy kedves tulajdonsága a Lightroomnak.

- **Utólagos finomhangolás a képtulajdonságokon képszerkesztőben:**

  Miután minden korrekciós fájl ki lett írva, érdemes a képeket importálni Lightroomba, hogy lássuk a módosítások eredményét, ne csak korrekciós értékeket, hiszen ezek a kész kép kontextusán kívül a laikus számára egészen semmitmondó számok. A Lightroom-on belül lehessen alakítani még a képeken, ha ez szükséges.

4. fejezet

Felhasznált technológiák, eszközök

Ebben a fejezetben az alkalmazáshoz felhasznált technológiák és fejlesztőeszközök rövid ismertetése található.

Python

Az alkalmazást python nyelven készítettem el, mert a legtöbb gépi tanulással, adatfeldolgozással és mély tanulással foglalkozó programkönyvtár erre a nyelvre létezik. A gépi tanulás és neurális hálók magas szintű programozási nyelve mára a python lett, ezért is volt célszerű ezt megtanulni. A pythonon kívül az R nyelvet szokták még használni, de egyre kevésbé elterjedt. Eddigi egyetemi gépi tanulás és mély tanulás kurzusokon mindenhol a python volt a választott nyelv. Ezen felül a keretrendszerek és könyvtárak is, amiket használok, a python nyelven a legteljesebbek. [27]

Visual Studio Code

Fejlesztőkörnyezetnek lokálisan a Visual Studio Code-ot választottam. Kipróbáltam a pycharmot is, ez IntelliJ python fejlesztőknek kínált környezete, de a Visual Studio Code jobb volt, az egyszerűbb konfigurálhatóság miatt.

Google Colab, jupyter

Mivel a neurális hálók tanítása nagyon számításigényes művelet, ezért a modellek tanítását nem lokálisan végeztem, hanem Google Colab-ban, egy jupyter notebookban. A Google Colab a Google felhőalapú környezete, amit korábbi gépi tanulás (machine learning) és mély tanulás (deep learning) témájú tárgyak alkalmával ismertem meg. Python és korlátozott mértékben shell kódot is lehet benne futtatni. A platform CUDA képes Nvidia GPU-t biztosít, ez a gyors tanuláshoz elengedhetetlen, de az ingyenes verzióban nem korlátlan a GPU hozzáférés, illetve nem választható meg a GPU típusa, de általában egy Nvidia Tesla T4-en sikerül tanítani. A Colab Pro fizetős csomagban egy adott mennyiségű számítási kapacitást ($\sim$ időt) lehet vásárolni ($\sim$ 50 óra), ez majdnem elégnek bizonyult, ennél többre volt szükség a projekthez.

A GPU használat előnyei:

A neurális hálózatok tanítása nagyon erőforrásigényes feladat, rengeteg mátrixművelet elvégzésére van szükség, a backpropagation (súlyok frissítése a hálóban) ebből áll. A GPU-k pedig pontosan ebben nagyon jók, gyorsan tudnak sok mátrixműveletet végezni.

A projekt egy korai állapotában, CPU-n futtatva a tanítást egy 70 képes adathalmazon nagyjából 8-9 másodperc volt egy tanulási ciklus (epoch) tanítási ideje. GPU-n

futtatva 40-50 milliszekundum volt ugyan ez. A 8 másodperces CPU-s és 40 milliszekundumos GPU-s időt véve is 200-szor gyorsabb volt a GPU-n tanítás egy tanulási ciklust nézve. Ez persze nagyban függ a Google Colab aktuális leterheltségétől, ugyanakkor jól szemlélteti a GPU-k hatékonyságát és megkerülhetetlenségét a mély tanulásban.

Google Drive

Az elkészített tanítási adathalmaz Google Drive-ból töltődik be, a Colab notebook innen olvassa be az adatokat. A tanítás közben generált statisztikák, grafikák és diagrammok, majd a betanult modellek szintén Drive-ra kerülnek mentésre. A Colab-nek ez is óriási előnye, hogy könnyen lehetett a Drive-val összekapcsolni.

TensorFlow, Keras

A tanításhoz TensorFlow-t és Keras-t használok. Ezek az könyvtárak kiegészítik egymást, magas absztrakciós szintet biztosítanak adatfeldolgozásra és mélytanulási modellek létrehozására, így nem teljesen az alapjaitól kell megvalósíti a különböző neuronokat, rétegeket, hibafüggvényeket és még sok más, alacsony szintű kódot. A TensorFlow felel a neurális háló GPU-n való tanításáért. [47]

plotly

A generált tanulási görbéket a plotly csomag segítségével rajzoltam ki, illetve tensorboardot használtam. Ez a TensorFlow modellek, modellkomponensek és tanítási görbék megjelenítésére szolgál.

rawpy, exiftool

A nyersképek és a korrekciós információk feldolgozásához a rawpy és exiftool csomagokat használtam. A rawpy a nyersképek beolvasásához és előfeldolgozásához, az exiftool pedig az .xmp fájlok feldolgozásához hasznos.

numpy

Ezen kívül numpy-t használtam, ez szintén egy gyakran használt könyvtár tudományos számításokhoz, valamint a többdimenziós tömbök, így képek kezeléséhez is.

pandas, datasets

Az adatok további feldolgozásában, hibaértékek megjelenítésében, valamint a képek TensorFlow számára is értelmezhető formára hozásában a pandas és a datasets csomag volt nagy segítségemre.

5. fejezet

Az alkalmazás fejlesztése

Az alkalmazás fejlesztése és a projekt az alábbi lépésekre bontható le, ezek a következő alfejezetekben kerülnek részletesebb kifejtésre:

- adatok előfeldolgozása,
- neurális hálók felépítése, modellek létrehozása,
- modellek konfigurálása,
- modellek tanítása,
- tanítás értékelése, grafikonok kirajzolása,
- modellek elmentése,
- prediktálás a modelleken,
- eredmények .xmp fájlokba írása, az eredeti nyersképek mellé,
- opcionálisan Lightroom meghívása, eredmények megtekintése.

5.1. Adatok

A tanulási adathalmaz .NEF, .CR2, vagy .ARW fájlformátumú nyersképekből áll, valamint mindegyikhez fájlhoz tartozik egy korrekciós .xmp fájl ugyan azzal a fájlnévvel a nyerskép mellett, ezáltal a korrekció és a nyers egyértelműen összerendelhető.

5.1.1. A nyers fájlformátumok (.NEF, .CR2, .ARW)

Közös jellemző ezekre a formátumokra, hogy feldolgozás nélküliek, a nyers szenzorból kiolvasott adat van beleírva. Jellemzően 12 vagy 14 biten tárolnak el egy színt, így nagyobb a képek dinamikatartománya, míg JPG-ben vagy és PNG-ben csak 8 bit tárolna egy színt. Mivel nincs előfeldolgozás, ezért például a fehéregyensúly, vagy a színárnyalatok teljesen szabadon állíthatók a retusálás során.

- .NEF (Nikon Electronic Format): A Nikon kamerák nyersformátuma, 12 vagy 14 biten tárol el egy színt. [34]
- .CR2 (Canon Raw 2): A Canon kamerák nyersformátuma, 14 biten tárol el legjobb minőségre állítva egy színt. [15]
- .ARW (Sony Alpha Raw Digital Camera Image): A Sony kamerák nyersformátuma, 14 biten tárol el egy színt. [14]

5.1.2. Az XMP fájlformátum

Az .xmp (Extensible Metadata Platform) az Adobe programok, mint a Lightroom és a Photoshop metaadat tárolásra használt formátuma, fejlesztését is az Adobe kezdte el. Szintaxisa alapján az .xml fájlokra hasonlít. Főleg képek nem destruktív szerkesztésénél szolgál a képen alkalmazandó változtatások tárolására, a retusálási értékek mentésére. Alapértelmezetten a Lightroom nem írja ki ezeket a fájlokat (bár ez beállítható), de kimenthetőek. Ekkor a Lightroom által meghatározott összes attribútum és metaadat kiírásra kerül, ilyen például a kamera és lencse típusa, fókusztávolság, fehéregyensúly, árnyalat, világosság, kontraszt, vibrálás, valamint még sok más paraméter. [16] [4]

5.2. Adatok feldolgozása

Az adatok feldolgozásával a *process_data.py* szkript foglalkozik.

A feldolgozás során első lépés volt az adatok megtisztítása. Érdemes manuálisan egy előválogatást tartani, ami után csak a jó képek maradnak benne az adathalmazban. Ha egy fájl nem a fentebb felsorolt formátumok valamelyikébe tartozott, az törlésre került a tanítási könyvtárból. A fájlnevek egy listába kerültek, a kiterjesztés és a név külön lett választva. Ha van .xmp és nyers képfájl is, akkor minden névből kettőnek kell lennie a listában. Ha ez valahol nem teljesül, azok a fájlok szintén törlésre kerülnek.

A következő lépésben, a könyvtáron végigiterálva megtörtént a nyersek és a hozzá tartozó .xmp-k beolvasása. A nyersek beolvasásához a rawpy csomagot használtam, ez beolvassa a nyersképet egy *rawpy.Rawpy* osztályba, amiből utána a *postprocess()* hívással [h*w*c] dimenziós numpy tömbként lehet letárolni a képet, ahol h a magasság, w a szélesség és c a csatornák száma. A *postprocess()* hívás sokféleképp paraméterezhető, ami ezek közül fontosabb lehet, hogy a feldolgozás során alkalmazza-e a kamera által mért és ajánlott fehéregyensúlyt, illetve hogy hány biten tároljon egy színt, jelenleg 16 biten és fehéregyensúly nélkül kerül mentésre a mátrixba a kép. Ez után a pixelek értékei [0, 1] tartományba lettek skálázva, majd átméretezve [133x200x3]-as méretre, ahol 133 pixel lesz a képek magassága, 200 pixel pedig a szélessége, 3 pedig a színcsatornák száma. Ez a méret kényelmesen kezelhető egy neurális háló számára, például az InceptionResNetV2 neurális háló, ami egy fejlett, képosztályozásra használt konvolúciós háló (residual blokkokkal és inception architektúrával) is [299x299x3] bemeneti dimenziókkal dolgozik. Még ebben a lépésben opcionálisan mentésre kerül a nyersek jpg-be konvertált verziója, majd az újraméretezett kép a képek listájához adódik. [23] [37]

Az .xmp fájlok feldolgozásához az exiftool csomagot választottam. Az exiftool beolvassa és parseolja az .xmp-t, majd a kiolvasott tageket kulcs-érték párok formájában adja vissza. Kicsit korábban elemezve ezeket az adatokat, megtalálható benne az 5 tanulni kívánt érték tag-je *("XMP:ColorTemperature", "XMP:Tint", "XMP:Exposure2012", "XMP:Contrast2012", "XMP:Vibrance")*, ezek egy listába kerülnek eltárolásra, majd ezeket az elvárt kimenetek listájához kell hozzáfűzni. [36]

A fentieket követően van két listák listája, ami azonos hosszúságú és párhuzamosan tartalmazza a bemeneti adatokat, illetve az elvárt kimeneteket. A két lista felcímkézés miatt a listák egy szótárba kerülnek, ez a későbbi feldolgozás során könnyít, majd egy *datasets.Dataset* osztályba konvertálódik. Ez a HuggingFace adathalmazok tárolására alkalmas osztálya, a projekt során ez bizonyult a legkönnyebben kezelhetőnek adathalmazok mentésére és betöltésére, a későbbiek folyamán ez volt az az adatformátum, amit a TensorFlow is fel tudott dolgozni. Ez után mentésre kerül az adathalmaz, opcionálisan az aktuális dátummal ellátva. [12]

Az adatok előfeldolgozása lokálisan kész, az adathalmazon végzett további műveletek már a Colab-ben történnek. Természetesen ez a lokális előfeldolgozás is nagyon számí-

tásigényes, nagyjából a kisebb, 70 kép előfeldolgozása nagyjából 10 percig tartott. Ennek ellenére megéri az erőfeszítés, mert egy 70 nyersképből (darabonként 24-30 MB, kameraszenzortól függően, legyen jelenleg 70 [db] * 24 [MB] = 1,6 [GB]) álló adathalmaz feldolgozva már csak $\sim$ 28 MB.

5.3. Neurális hálók implementálása

Az alkalmazás fejlesztésének ezen része már Google Colab-ben folytatódott.

Az első blokkban a szükséges csomagok importálása történik, ezek többnyire a TensorFlow és a Keras különféle rétegei, modelljei és függvényei, amik az adatok betöltéséhez és a neurális hálók építéséhez kellenek. Inicializálásra kerül a tensorboard, ami egy diagnosztikai eszköz, többek között a tanulási folyamatról ad grafikonokat és a neurális háló felépítését lehet benne grafikusan megtekinteni.

Ez után a korábban exportált adathalmaz beolvasását végzem el Google Driveból, majd ez szétválasztásra kerül tanulási és tesztelési adathalmazokra, ezeket pedig tovább bontom a képekre (a bemenetre) és célértékekre (az elvárt kimenetre). A tanulási és tesztelési halmazok külön definiálására azért van szükség, mert a tanítás minőségéről úgy kaphatunk pontos eredményt, ha tanultakat új, még ismeretlen bemeneten értékeljük. A valóságban is így lehet jól tudást, tanulási minőséget mérni, ez a neurális hálók esetében sincs másképp. Ez után minden képez, ami mátrixként van tárolva, megvan egy listában az 5 jellemző, mint elvárt kimenet.

Az adatok importálása után az adatok augmentálása következik. Ez egy technika, ami az adathalmazból kiindulva új mintákat generál a képek változtatásával, ezáltal bővítve az halmazt. Így a hálózat nagyobb adathalmazon tanul és robusztusabb lesz. Az augmentáció előfeldolgozó (preprocessing) rétegekkel valósul meg. Ezek közül a *RandomFlip*-et és a *RandomRotation*-t használtam fel, mert ezek nem változtatják olyan mértékben a képet, hogy az a végeredményt komolyan befolyásolná. Ezek a rétegek véletlenszerűen horizontálisan tükrözik és forgatják bizonyos szögtartományok között a bemeneti képet. Mivel alapvetően kevés adat állt rendelkezésre, ezért az augmentált és eredeti képek is a tanítási halmazba kerültek.

Mivel a jellemzők közül néhány egészen eltérő értéktartományban van (a fehéregyensúly ezres nagyságrendű, míg a többi jellemző tizes nagyságrendben található), ezért 5 külön neurális háló készül, ezt azt jelenti, hogy az elvárt kimeneteket szét kell választani az 5 hálónak megfelelően.

A projekt során Keras-t és TensorFlow-t használtam, ezek segítségével építettem fel a neurális hálókat. A következő rész a felhasznált osztályokról, korábban ismertetett rétegekről fog szólni.

Felhasznált Keras rétegek:

- *Dense:* Teljesen normális fully connected réteg a neurális hálóban. A rétegben lévő neuronok száma és az aktivációs függvény fajtája, ami ennél a rétegnél fontosabb paraméter és specifikálni lehet.
- *Dropout:* Egy előre definiált valószínűséggel tanítási időben véletlenszerűen kihagy kapcsolatokat a rétegek között, ezzel segítve, hogy a hálózat ne tanuljon túl (overfitting). Alkalmazási időben már a véletlenszerűen kihagyott kapcsolatok is részt vesznek a predikcióban. Meg lehet határozni, hogy a kapcsolatok hányadrésze legyen kihagyva.
- *Activation:* Réteg az aktivációs függvénynek.

- *Flatten:* Sorosító réteg, ami kilapítja a bemenetet és egy vektort hoz létre belőle. A bemenet konvolúciós hálók esetén célszerűen mátrix formában van, például lehet ez egy aktivációs térkép, ami egy konvolúciós réteg kimenete. Nem tartalmaz tanítható paramétereket, hiszen csak egy egyszerű átdimenzionálási művelet.

- *Conv2D:* Konvolúciós réteg, mintavételezésre szolgál. Megadható a konvolúciós filterek száma, illetve a filterek (kernel) mérete, illetve a réteg kimenetéhez az aktivációs függvény. Amennyiben első rétegként van a hálóban, úgy a bemeneti méretet is meg kell adni. Leggyakrabban 3x3-as kerneleket használnak. 2x2-es, vagy 4x4-es kernelek használata nem ajánlott, mert ezek szisztematikusan osztják el a bemeneti kép pixeleit a kimeneti pixel körül. Régebben például az AlexNet-nél használtak nagyobb, 5x5, 7x7, vagy 11x11-es kerneleket. Később kiderült, hogy a nagyobb rétegeket ki lehet váltani kisebb rétegek egymás utáni alkalmazásával. A több réteg bonyolultabb összefüggések felismerésére képes, ezért előnyösebb több kisebb filtert alkalmazni. [35]

- *MaxPooling2D:* Maximum összevonó réteg, itt a kernel méretét kell megadni. Régebben használtak nagyobb filtereket, ma a 2x2-es és a 3x3-as méret a legelterjedtebb. Ez a réteg az előző réteg méretét csökkenti és nyeri ki belőle az adott területről domináns információt.

- *GlobalAveragePooling2D:* Globális összevonó réteg, az egész aktivációs térkép számtani közepét veszi, így nincs paraméterezése, egy értéket ad tovább. Sorosító réteg helyett célszerű használni.

- *Add:* Ez a réteg több réteg összevonására szolgál, amiknek azonos a kimeneti dimenziója. A residual connection-öknél kerültek beépítésre, ahol konvolúciós rétegeket és a bemenetet összegzi. Paraméterei az összeadandó rétegek egy listába rendezve.

- *BatchNormalization:* Normalizálja az adatokat egy batch-en belül. Re-centering-et és re-scaling-et hajt végre.

- *RandomFlip:* Adataugmentációs réteg, a bemeneti képet véletlenszerűen tükrözi, beállítástól függően függőlegesen és/vagy vízszintesen.

- *RandomRotation:* Szintén adataugmentációs réteg, a bemeneti képen végez véletlenszerű forgatást meghatározható szögtartományokban. [45]

A modelleket felépítő függvények deklarációja a könnyebb kezelhetőség érdekében egyformák, eltérések csak a paraméterek értékében, illetve a függvénynevekben vannak.

```
def buildNN_X(input_shape: tuple, output_shape: int,
          showModel: bool = False, with_dropuot: bool = True,
          with_normalization: bool = True, name: str = 'buildNN') -> Sequential:
```

Minden háló első rétegének meg kell adni a bemeneti és kimeneti dimenziót, erre szolgál az *input_shape* (jelenleg: [133, 200, 3]) és *output_shape* (jelenleg: 1) paraméter.

A *showModel* kapcsoló a modellfelépítés kiírását és képként való mentését engedélyezi a modellkonstrukció végén.

A *with_dropout* a dropout rétegek neurális hálóhoz adását kapcsolja ki/be, alapbeállításban be van kapcsolva. A konkrét dropout értékek hálóról hálóra változhatnak, kezdetben egységesen 0.1, vagyis a neuronok közötti kapcsolatok tizede lesz véletlenszerűen tanítás alatt eldobva.

A *with_normalization* kapcsoló a BatchNormalization rétegeket kapcsolja, ezek az adott batchet re-centerelik és újraskálázzák. (re-centering and re-scaling). Bár még nincs

teljesen megértve, mint koncepció, de elméletileg segíti a modell tanulását, stabilabb és gyorsabban tanul a modell batch normalizálással, mint nélküle. [20] [8]

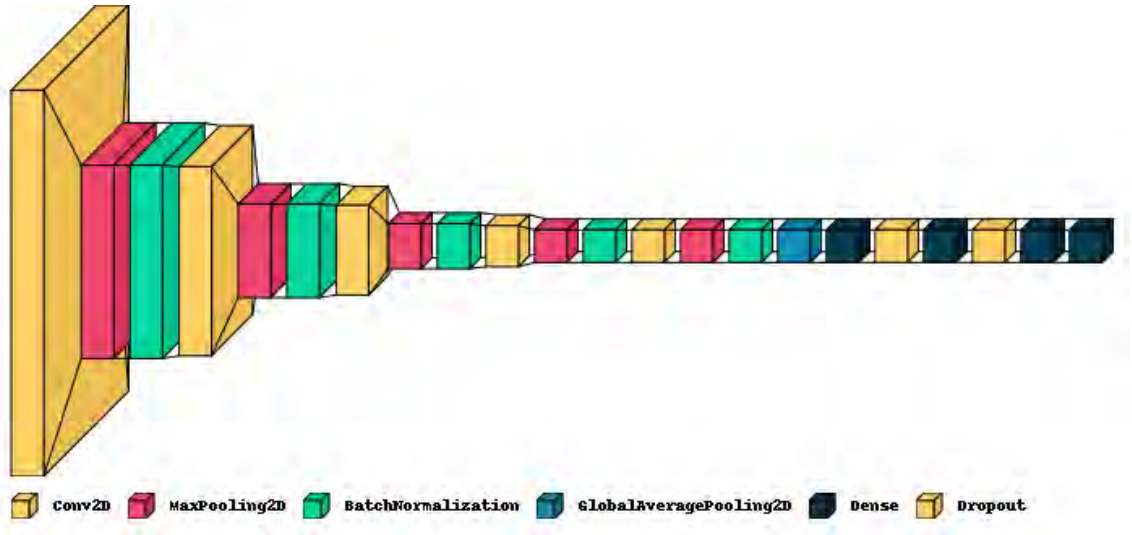
A *name* paraméter értelemszerűen a modell neve lesz.

A modellépítő függvények üres Sequential (Modelből specializált osztály) vagy Model példányra kezdik el építeni a rétegeket és ilyen típussal is térnek vissza. Alapvetően Sequential-t és a Keras szekvenciális API-ját használtam az egyszerűbb deklarálási mód miatt. Model-re akkor volt szükség, amikor funkcionális API-t kellett használni a residual connection-ök miatt, a Sequential nem engedi a rétegek szétágazását és összevonását, csak a szekvenciális építkezést. [48]

5.4. Modellek létrehozása

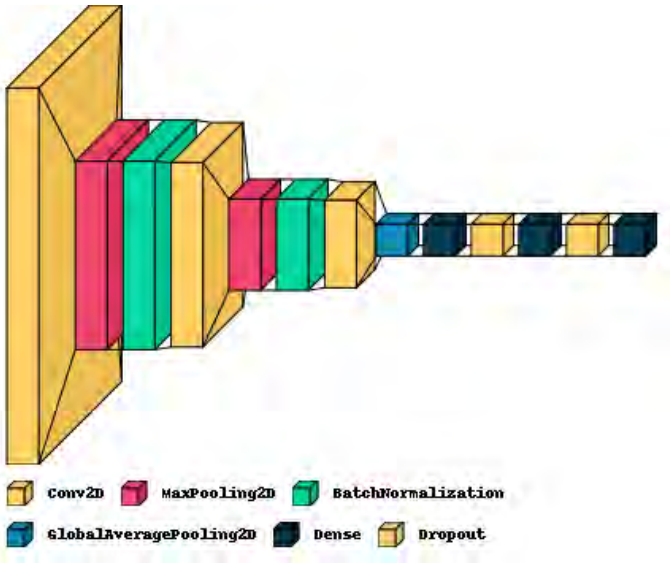
Mindegyik modellnek van egy saját modellépítő függvénye, ami létrehozza az adott hálózatot.

buildNN_init: egy egyszerű mély konvolúciós hálót hoz létre. Felépítése a 5.1 ábrán látható.



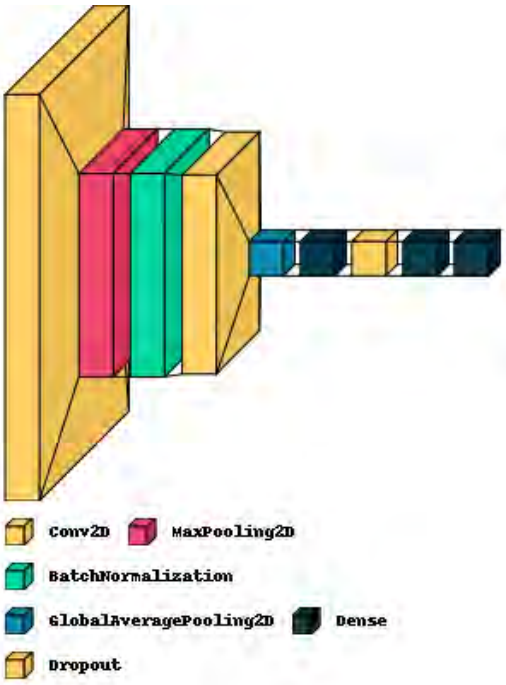
5.1. ábra. A *buildNN_init* architektúra felépítése

buildNN_s: Az előző háló egyszerűbb verziója, egyel kevesebb Conv-Relu-MaxPool-BatchNorm blokkal, kisebb filterméretekkel, valamint egyel kevesebb teljesen csatolt (fully connected) réteggel.



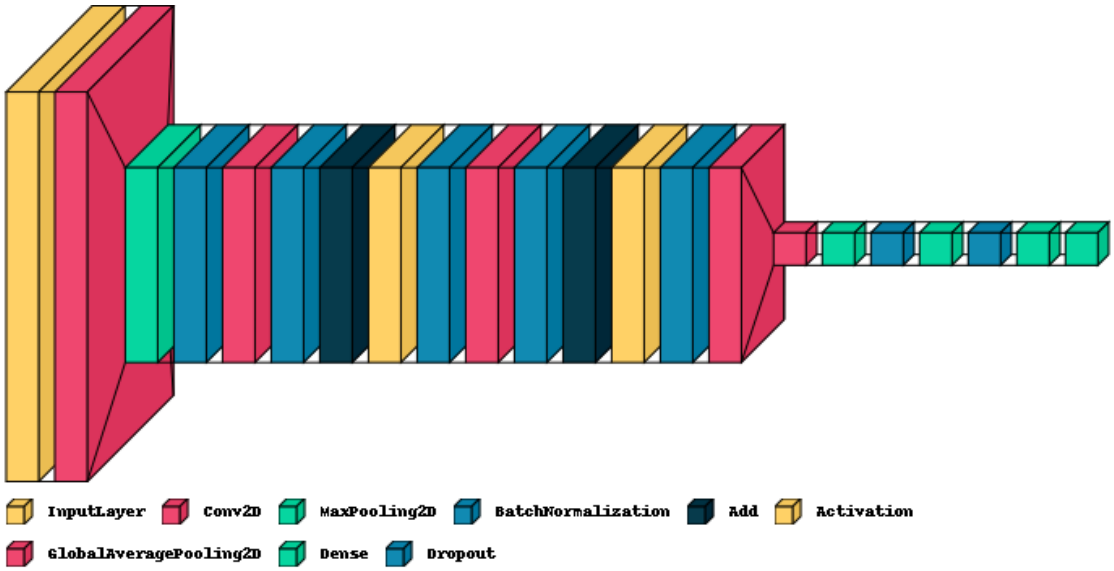
5.2. ábra. A *buildNN_s* architektúra felépítése

buildNN_xs: A *buildNN_s*-hez képest kisebb háló, kevesebb rétegből áll.



5.3. ábra. A *buildNN_xs* modell felépítése

buildNN_RCNN: A *buildNN_init* alapjain indult háló, de a konvolúciós rétegek között van két visszacsatolt kapcsolat (residual connection).

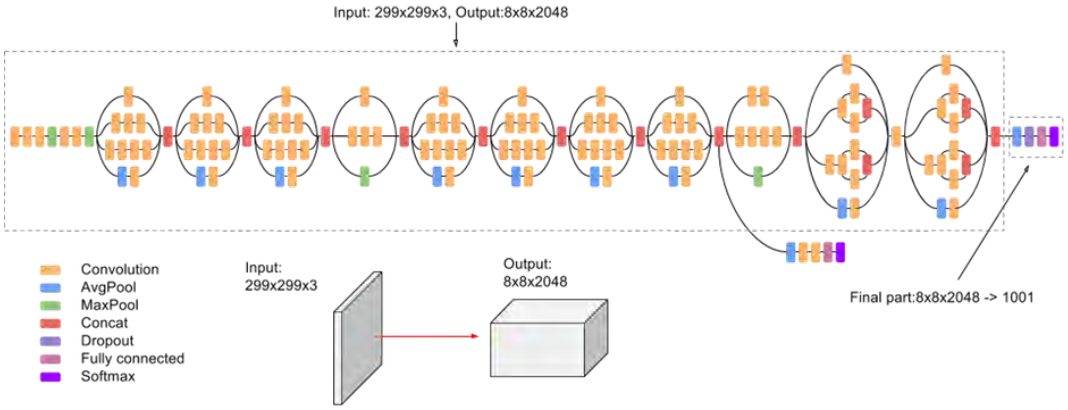


5.4. ábra. A *buildNN_RCNN* modell felépítése

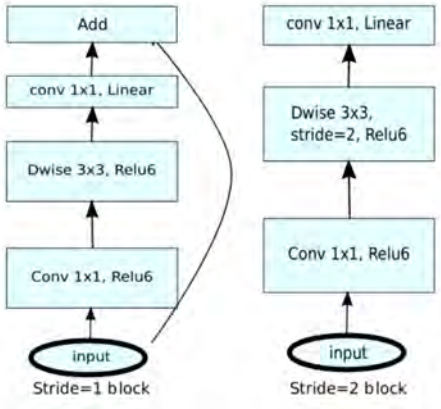
buildNN_pretrained: Egy előre tanított hálózat konvolúciós rétegeit veszi alapul, erre ülteti rá a GlobalAveragePooling2D és a fully connected rétegeket. Transfer learninget valósít meg. Az alábbi ábrán a függvény által felépített háló látható, a Functional rész az alapháló, ennek belső részleteit sajnos nem tudta a vizualizációhoz használt eszköz megjeleníteni. A MobileNetV2 és InceptionV3 hálókkal kísérleteztem mint alaphálók. Ezek nem bizonyultak elég hatékony megoldásnak, túl komplexek voltak, mint alapmodell.

5.5. ábra. A *buildNN_pretrained* modell felépítése

A bemenet természetesen újra lett méretezve, hiszen ezek a hálók alapvetően nem 133x200 felbontású képeken lettek tanítva. A transfer learning miatt az utolsó osztályozáshoz használt rétegek kimaradtak, helyette a fentebbi ábrán látható rétegek kerültek.



5.6. ábra. Az InceptionV3 architektúrája [22]



5.7. ábra. A MobileNetV2 architektúrája [30]

5.5. Transfer learning

Ha kevés adat áll rendelkezésre és van egy hasonló probléma, amire már van egy hatékony, nagyobb adathalmazon betanított neurális hálózat, akkor érdemes a betanított hálózat rétegeit kiindulásként felhasználni. Ez úgy valósítható meg, hogy az előre betanított hálózat rétegeit befagyasztják, ezek a rétegek a tanulás első fázisában nem taníthatók. Ezután a háló tetejére saját rétegeket csatlakoztatunk, ezek viszont taníthatók. A tanulás első fázisában így tanítják a hálót a tanulási adathalmazon. Amikor már elég stabil ez a háló, akkor az alaphálót taníthatóra állítjuk és így is hagyjuk tanulni a modellt, így az alsóbb rétegek finomhangolása is megtörténik.

A gyakorlatban a transfer learning nem kivételes megközelítés, ez a standard, mert általában nincs elég adat és idő az igazán mély modellek teljes betanítására.

5.6. Modellek specifikálása

Ez után az 5 modell ténylegesen példányosításra kerül, hozzájuk kell rendelni még a hibafüggvényeket és optimalizálókat. Ekkor kezdődik a hosszas modellválasztási folyamat, ahol a modellparaméterek részben kézi hangolására van szükség.

Ilyen modellparaméterek például a:

- neuronok száma (hány neuron van egy rétegben, mekkora a konvolúciós, összevonó filter),
- rétegek száma (hány réteg van egymás után, hány konvolúciós filtert tartalmaz a háló),
- dropout alkalmazása (van-e a hálóban dropout réteg, milyen rátával van),
- batch size (egyszerre hány mintára számol a háló hibát),
- tanulási ciklusok (epochs) (hányszor megy át a teljes adathalmaz a hálón, vagyis hány cikluson keresztül tanul a háló),
- háló architektúrája (van e visszacsatolt kapcsolat (residual connection) a hálóban, vagy egyéb elágazás; milyen az összevonó réteg; esetleg egy előre betanított háló tetejére kerül pár teljesen csatolt (fully connected) réteg (transfer learning)),
- aktivációs függvények (milyen aktivációs függvények vannak a hálóban),

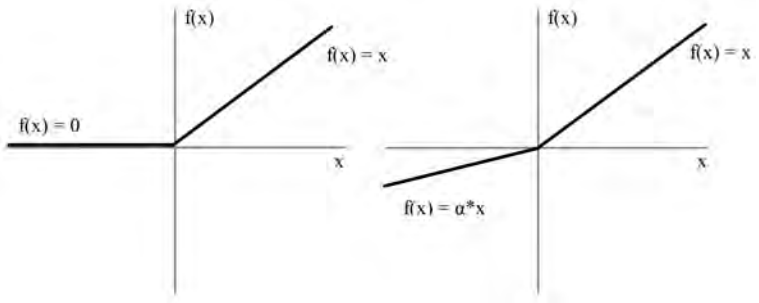
- hibafüggvények (több fajta van, fontos elem a paraméterfrissítésnél),
- optimalizálók (tanulási rátát befolyásol, azaz a tanulás sebességét).

A következő rész a kipróbált és felhasznált aktivációs függvényekkel (activation functions), hibafüggvényekkel (loss functions) és optimalizálókkal (optimizers) foglalkozik.

5.6.1. Aktivációs függvények (activation functions)

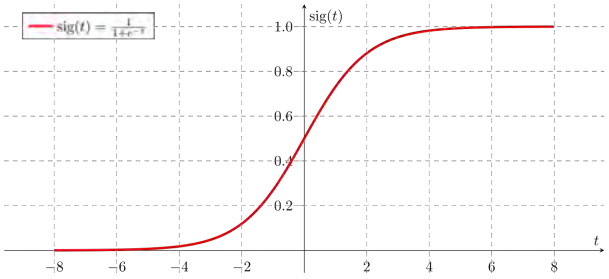
A neuron kimenetét szabályozzák, a gyakorlatban az alábbi két aktivációs függvény terjedt el regressziós problémákhoz egyszerűségük miatt.

- ReLU (Rectified Linear Unit): Ha a bemenet nagyobb, mint nulla, akkor a bemenet, egyébként nulla.
- Leaky ReLU: Ha a bemenet nagyobb, mint nulla, akkor a bemenet, egyébként az α paraméterrel szorzott bemenet.



5.8. ábra. ReLU és Leaky ReLU aktivációs függvények képei [43]

Az régebbi neurális hálók sokat használták aktivációs függvényként a sigmoidot. Ezt lehetne használni, de könnyen szaturálódik, ezért nem célszerű. Szaturálódás, amikor egy nagy x-beli eltérés esetén a függvény értékében kicsi az eltérés, ez az alábbi ábrán jól látható a nagyon nagy, illetve nagyon kicsiny x értékek esetében. A gyakorlatban a legtöbb regressziós problémánál a ReLU-t, vagy valamilyen változatát alkalmazzák gyors számíthatósága és egyszerűsége miatt.



5.9. ábra. A sigmoid függvény képe [44]

5.6.2. Hibafüggvények (loss functions)

A hibafüggvények a prediktált kimenet és az elvárt kimenet alapján számolnak egy értéket, ami a backpropagation során visszakerül a hálóba, ez alapján történik a súlyok frissítése.

- Mean Square Error (MSE): Hiba négyzetének átlaga, ez a legáltalánosabb regressziós problémák esetén.
- Mean Absolute Error (MAE): A hiba abszolútértékének átlaga, óvatosabb hibafüggvény mint a MSE, mert a gradiens állandó, illetve kevésbé érzékeny a kiugró adatokra.
- Huber loss: A MAE és a MSE kombinációja. Egy bizonyos hibaértéken belül MSE-t, azon felül MAE-t megvalósító hibafüggvény.

5.6.3. Optimalizálók (Optimizers)

Az optimalizálók olyan algoritmusok, amik a súlyok frissítésének helyes mértékéért felelnek, azaz a háló konvergenciájáért felelnek. A háló tanítása során cél, hogy a súlyok úgy frissüljenek, hogy a hibafüggvény értéke csökkenjen, a hiba minimalizálódjon. A hibafüggvény megadja a háló hibáját az adott mintára, ezt az értéket a súlyok szerint deriválva a frissítés mértékét lehet kiszámolni, ez egy gradiens. A tanulás gyorsítása érdekében ezeket a gradienseket még egy tanulási rátának nevezett paraméterrel szoroznak meg. Az optimalizáló algoritmusok ezt a tanulási rátát vagy rátákat módosítják, adott esetben adaptívan a gradiens nagyságához, vagy a dimenzió súlyfrissítéséhez képest.

Az alábbi optimalizálókat használtam fel a neurális hálókhoz, ezek a *tf.Keras.optimizers* csomagban találhatók:

- RMSProp (Root Mean Square Propagation): Adaptívan frissíti a tanulási rátát (learning rate), a korábbi és jelenleg kiszámított gradiensek súlyozott átlagának gyökével súlyozza a tanulási rátát. [39]
- Adagrad (Adaptive Gradient Descent): Minden paraméterhez egyéni tanulási ráta (learning rate) tartozik, amely a frissítések gyakoriságával, illetve a gradiensek nagyságától függően csökken. Minden lépésben figyelembe veszi az összes korábbi gradienst. [2]
- Adadelta (adaptive delta): Az Adagrad csúszóablakos változata. Jobb, mint az Adagrad, mert akkor is képes még frissíteni a paramétereket, amikor az Adagrad már nem. Ennek oka a Adagradnál alkalmazott gradiensakkumulálás. [1]
- Adam (Adaptive Moment Estimation): Az RMSProp és a Momentum ötleteit kombinálja. Minden paraméterhez saját adaptív tanulási rátát számol és figyelembe veszi a korábbi gradienseket, azok négyzetét, korrigálja ezeket és ez alapján frissíti a paramétereket. [3] [31]

Az SGD (Stochastic Gradient Descent) mint alap optimalizáló használata is felmerült, ám még az első hálók tanítása során kiderült, hogy nem elég hatékony és nagyon nagy kilengések vannak a tanulási görbében. A fentebbi optimalizálók sokkal jobban teljesítettek. Az SGD optimalizáló mindig véletlenszerűen választ egy mintát, majd ennek a hibából számolt gradiensét szorozza meg egy előre meghatározott α paraméterrel, majd ezt kivonja a súlyokból. Ennek egy továbbfejlesztése, amikor véletlenszerűen választ ki több mintát (batch-et), és a gradiensek átlagával frissíti a súlyokat. [32] [42]

A következő részben a modellek elemzéséről és a modellek finomhangolásáról lesz szó.

6. fejezet

Modellek leírása, eredményeik elemzése, modellválasztás

Ebben a fejezetben a modellek kiválasztását és finomhangolását tárgyalom. A modell selection-nek nevezett feladat és problémakör a legnehezebb bármilyen gépi tanulási problémánál, hiszen számos paraméter optimális kombinációját kell megtalálni, amikkel a háló a lehető legjobban teljesít a feladaton. Alapvetően két problémára kell figyelni és a paramétereket ezek szerint beállítani: a túltanulás (overfitting) és az alultanulás (underfitting). Bármelyik probléma esetén a háló nem elég jó, de a megnevezett problémák hiánya sem jelenti azt, hogy a háló tökéletes a feladatra. Előfordulhat, hogy háló bár tanul, így se képes elég jó eredményt produkálni, vagy csak túlzottak a feltételekhez képest a támasztott elvárások. Ezek miatt nehéz a modellválasztás feladata.

6.1. Overfitting, underfitting

Overfitting (túltanulás):

A modell túlságosan megtanulja a tanulási adathalmazt, nem elég általános a tanulás ahhoz, hogy jó eredményt érjen el a modell a validációs adathalmazon.

Underfitting (alultanulás):

A modell nem elég komplex ahhoz, hogy megtanulja az adatokban fellelhető mintázatokat és képtelen elég jó eredményt elérni a validációs halmazon. Ezen kívül a tanulási görbén a tanulási halmaz eredményeinél észrevehető, hogy a modell nem képes tovább javulni és alacsonyabb hibával prediktálni. [10]

6.2. Adathalmazok

A tanuláshoz a következő adatokat használtam:

Konstanz adathalmaz:

Az adathalmaz egy nyári, konstanzi kirándulás képeiből áll, összesen 56 kép a tanulási halmaz. A validációs halmazban 12 kép található. Jellemzően nappali, jó megvilágítású képek.

Szakestély adathalmaz:

Az egyik Villanykari Szakestély képeiből álló adathalmaz. 113 képből áll, 90 kép a tanulási halmaz, 23 kép marad a validációs halmaz. Fényszegényebb képek.

Mixed adathalmaz:

Képek a fenti két adathalmazból fele-fele arányban.

6.3. Modellválasztás

A modellválasztást a fent definiált 5 architektúra vizsgálatával kezdtem egy alap paraméterezéssel a Mixed adathalmazon. Minden háló 200 tanulási ciklust hajt végre, ha early stopping miatt meg nem áll hamarabb.

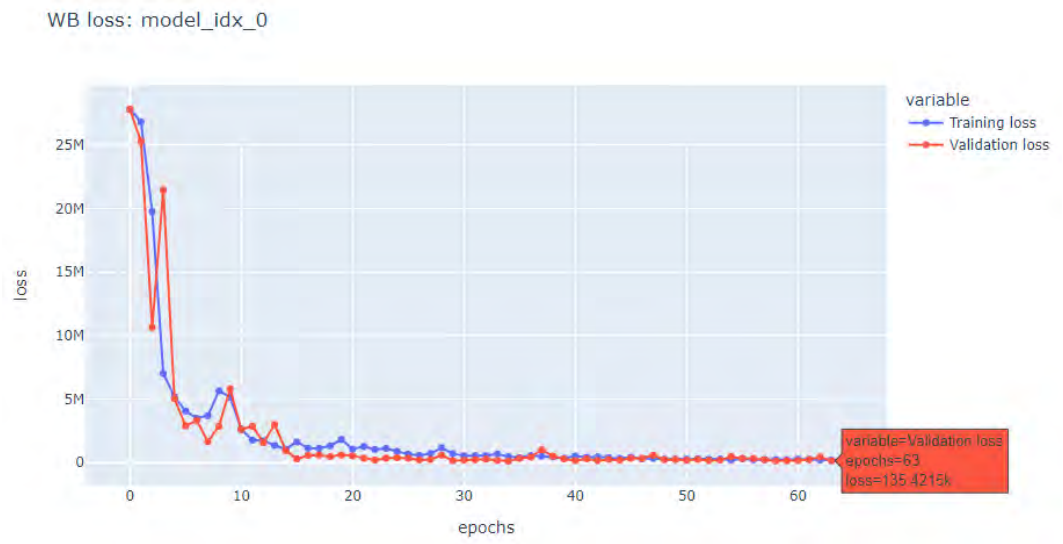
Early Stopping:

Az early stopping egy heurisztika a túltanulás megelőzésére. A módszer lényege, hogy a validációs hibákat és epochonként a súlyokat feljegyzi az algoritmus és ez alapján dönt a tanítás folytatásáról vagy befejezéséről. Ha megadott számú epochig nem javul a háló validációs hibája a korábbi legjobbnál, akkor leáll a tanulás és az addigi legjobb eredmény súlyait alkalmazza a későbbiek folyamán a modell.

A batch méret 4 mintából áll, ez azt jelenti, hogy 4 képenként történik súlyfrissítés. A hálók Adam optimalizálót (optimizer) és MSE hibafüggvényt (loss function) használnak. Első lépésben 5 háló egyetlen kimeneti paraméterre, a fehéregyensúlyra tanul, így segítve a különböző modellek közötti teljesítménybeli különbségek észrevételét és kiértékelését.

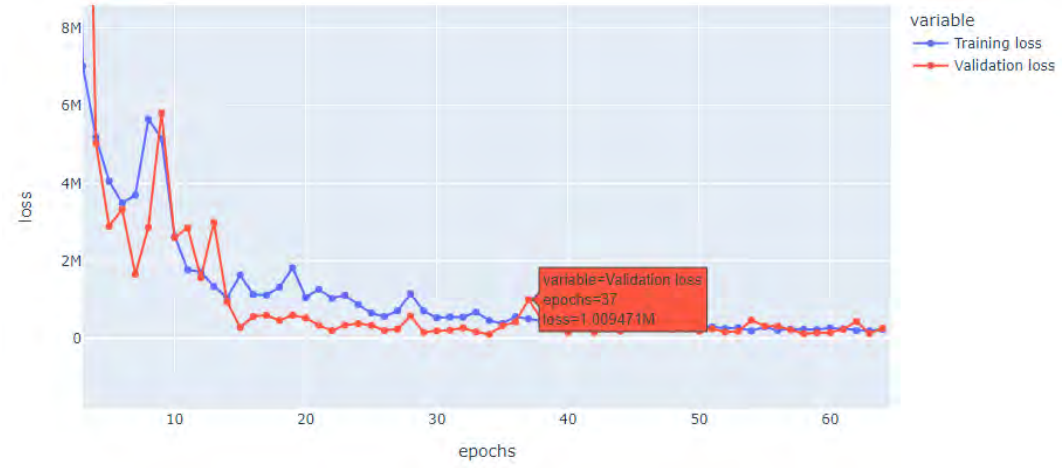
6.3.1. Szekvenciális konvolúciós modellek

Az alábbi 6.1 ábrán az első, *buildNN_init*-tel felépített háló tanulási görbéje látható. A grafikonokon is az x tengelyen a tanulási ciklusok (epochs) száma, az y tengelyen a számolt hiba értéke látszik, mind a tanulási (training set), mint a validációs halmazon (validation set). A hibaértékek milliós nagyságrendje a fehéregyensúly esetén nem meglepő, a fehéregyensúly mérőszáma az ezres nagyságrendben van. Ha a háló ebben a nagyságrendben téved, úgy a négyzetre emelés után a milliós nagyságrend teljesen indokolt.



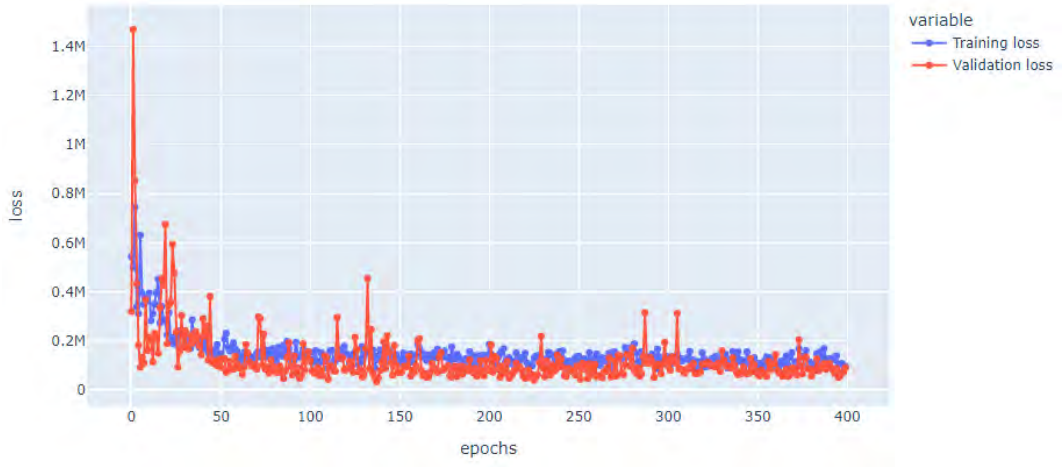
6.1. ábra. A *buildNN_init* háló tanulási görbéje

A fenti 6.1 grafikonon a tanulási hiba nagyjából folyamatosan csökken, a validációs hiba viszont a tanulási görbét nagyítva kevésbé stabil értékeket mutat, de alapvetően csökken. Ígéretes, hogy a ciklusok végére kezd stabilizálódni a háló, ugyanakkor a tanulás elején egészen tág intervallumban mozognak az értékek.

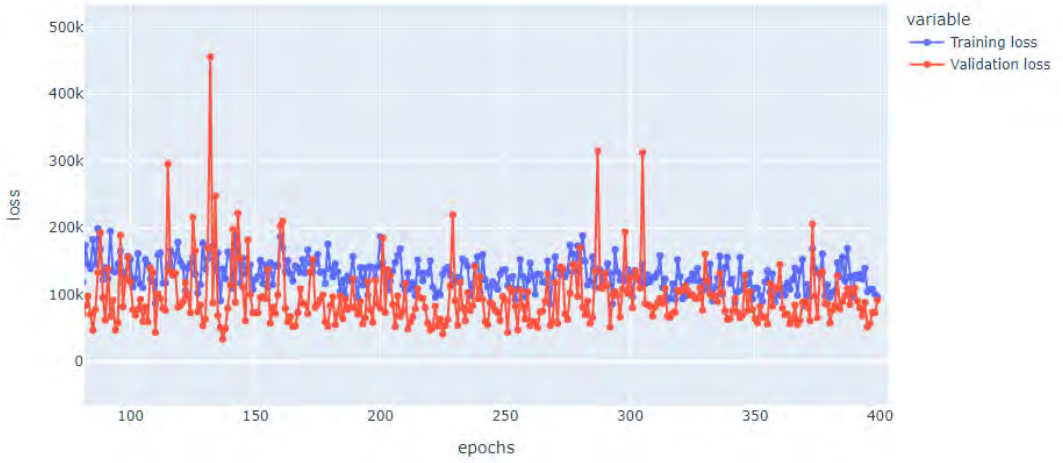


6.2. ábra. A *buildNN_init* háló tanulásának nagyított grafikonja

Az early stopping megfogja a hálót egy optimális paraméterezésnél, ahol a validációs hiba kicsi.



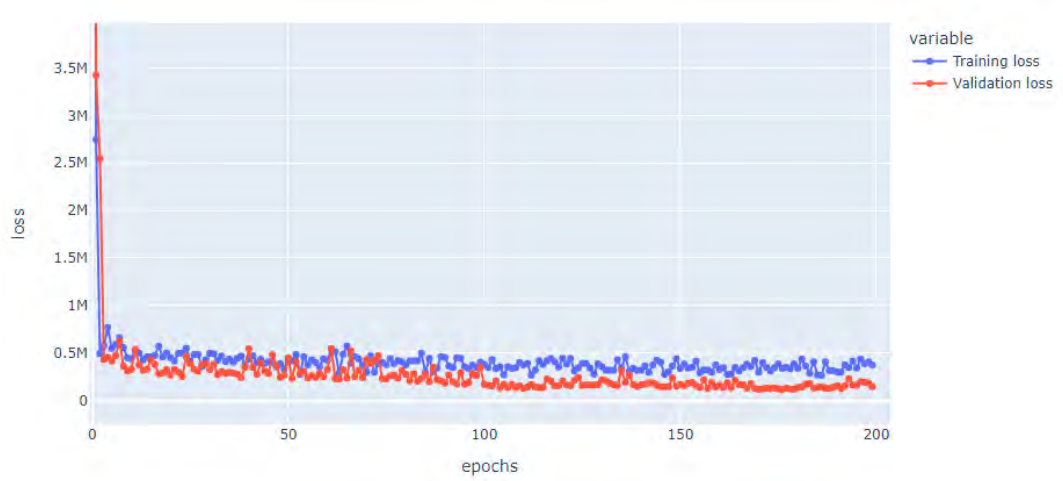
6.3. ábra. A *buildNN_init* háló tanulása early stopping nélkül



6.4. ábra. A *buildNN_init* háló hibagrafikonja nagyítva, early stopping nélkül

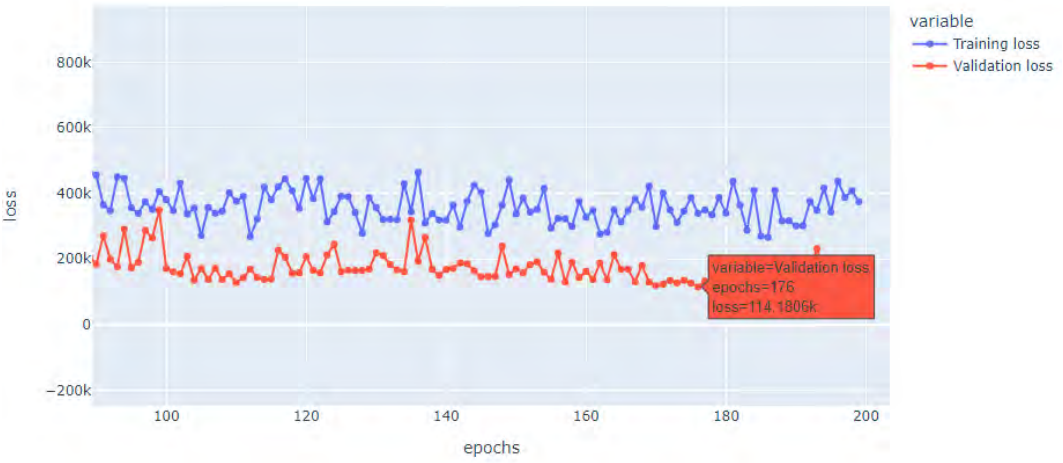
A fenti 6.3 ábrán a háló early stopping nélküli tanítási folyamata látható. Feltűnő, hogy mennyire kiugró validációs hiba értékek is előfordulnak, alapvetően viszont a 130.000-res hibaértékeknél alacsonyabb, gyakran 70-80 ezres hibát lehet tapasztalni anélkül, hogy az túltanulás (overfitting) jelentkezne. A hosszabb tanulás során (200 helyett 400 epoch) a tanulási és validációs hiba is csökkent még, vagyis az early stopping-ot érdemes ennél a hálónál máshogy paraméterezni (6.4 ábra). Szerencsére az optimalizáló és a hibafüggvény is jónak tűnik, így ezeken egyelőre nem szükséges változtatni.

A következő háló a *buildNN_s* függvénnyel lett létrehozva, ez kevesebb konvolúciós (convolutional) és teljesen kapcsolt rétegből (fully connected layer) áll, mint a *buildNN_init* által létrehozott háló, illetve kevesebb neuron van rétegenként. Tanulási görbéje következőképp néz ki:



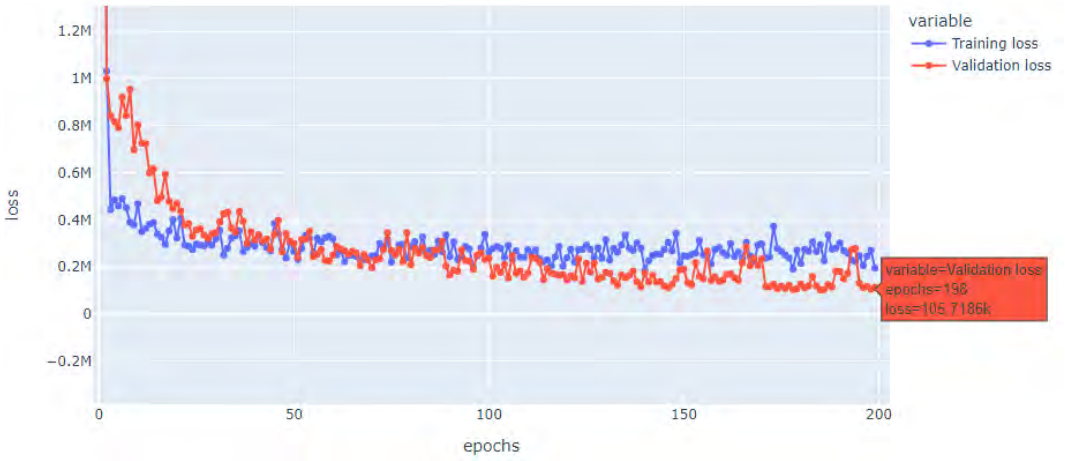
6.5. ábra. A *buildNN_s* modell tanulási görbéje

A fentebbi (6.5) grafikonon sokkal bíztatóbb eredményeket lehet megfigyelni, a háló stabilabban konvergált, nincs már annyi kiugró érték a validációs hibáknál. Az alábbi 6.7 grafikonon azonban jól látszik, hogy nem ért el annyira jó hibaértékeket, mint az előző háló.



6.6. ábra. A *buildNN_s* háló tanulási görbéje. Ez sajnos nem ér el 200 ciklus alatt annyira jó eredményt, mint az előző háló

A harmadik modell, ami a *buildNN_xs*-függvénnyel lett létrehozva, hasonlóan jó eredményeket mutat. Ez a háló kevesebb rétegből áll, mint a *buildNN_s*, de hasonló hibaértékeket képes produkálni.



6.7. ábra. A *buildNN_xs* tanulási görbéje

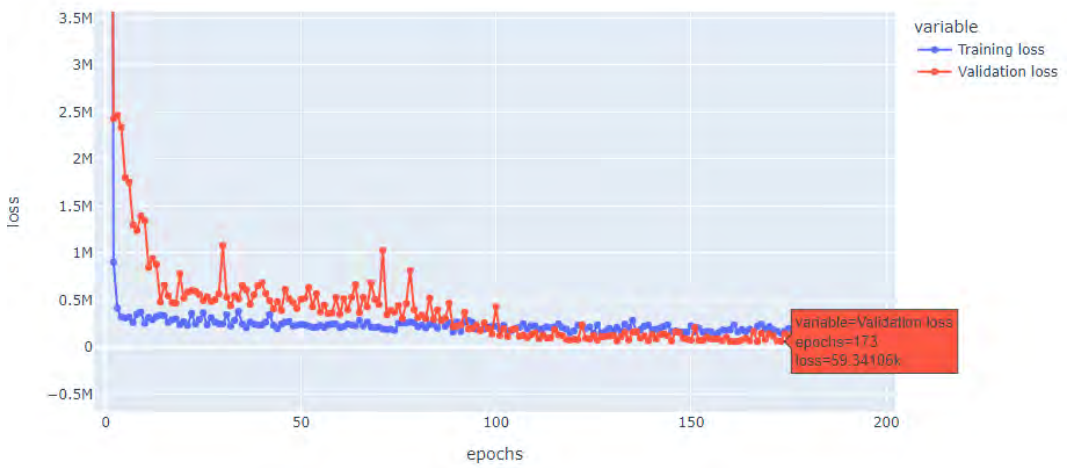
6.3.2. Visszacsatolt háló

A következő háló a *buildNN_RCNN* szüleménye. Ennek a hálónak a visszacsatolt kapcsolatok (residual connection) a különlegessége. Felépítésben talán leginkább az első, *buildNN_init* hálóhoz hasonlít, de a konvolúciós rétegek kevesebb filterből állnak és a filterek között összeadó rétegek is vannak, amik pár konvolúciós réteggel korábbi aktivációs térképed adnak az aktuális aktivációs térképhez. Ez a technika a háló mélyebb rétegeiből biztosít információt a későbbi rétegeknek, ezzel a vanishing gradients/exploding gradients problémát oldja meg.

A vanishing/exploding gradients probléma mély neurális hálóknál fordul elő a nagy mélységbe visszapropagálandó súlyfrissítéseknél. Ilyenkor a visszapropagálás (backpropagation) során a sok szorzás miatt a gradiens vagy 0 lesz, ha kezdetekben egynél kisebb volt, vagy végtelen, ha kezdetben egynél nagyobb volt. Egyik eset sem optimális, mert így nem tudnak a mélyebb rétegek tanulni.

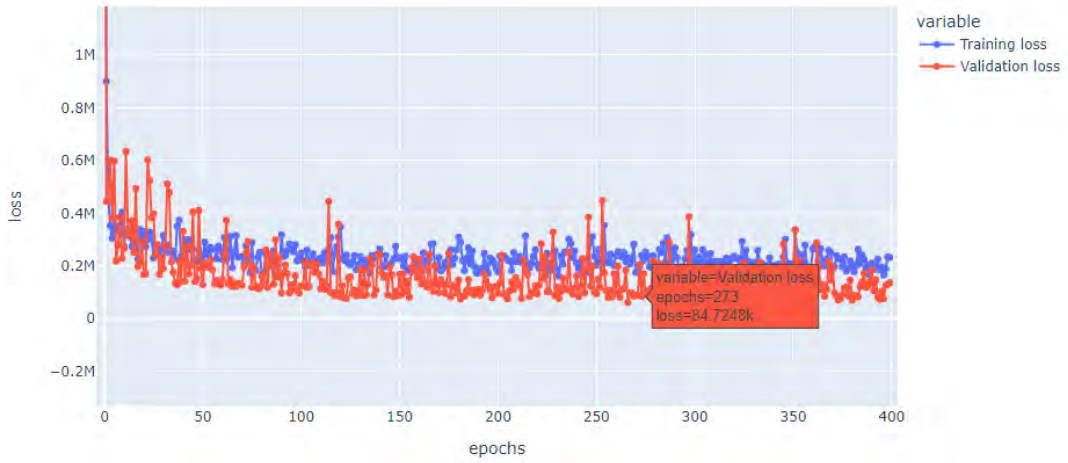
A visszacsatolt kapcsolatok (residual connection) miatt több neurális hálót szimulál egy architektúra, mivel a gradiensek frissítésének több útja van, így a háttérben nagyjából úgy viselkedik, mint több háló.

A teljes tanulási görbe alább a 6.8 ábrán látható. Nagyon ígéretes az eddigieknél alacsonyabb validációs hiba, elképzelhető, hogy érdemes lenne ezt a hálót tovább tanítani.



6.8. ábra. A residual connectionnel ellátott *buildNN_RCNN* modell tanulási görbéje

A hálót az előzőhöz képest kétszer tovább tanítva se lett jobb az eredmény, ez az architektúra valószínűleg ennyit tud, ugyanakkor így is további fejlesztésre érdemes lehet kiválasztani.



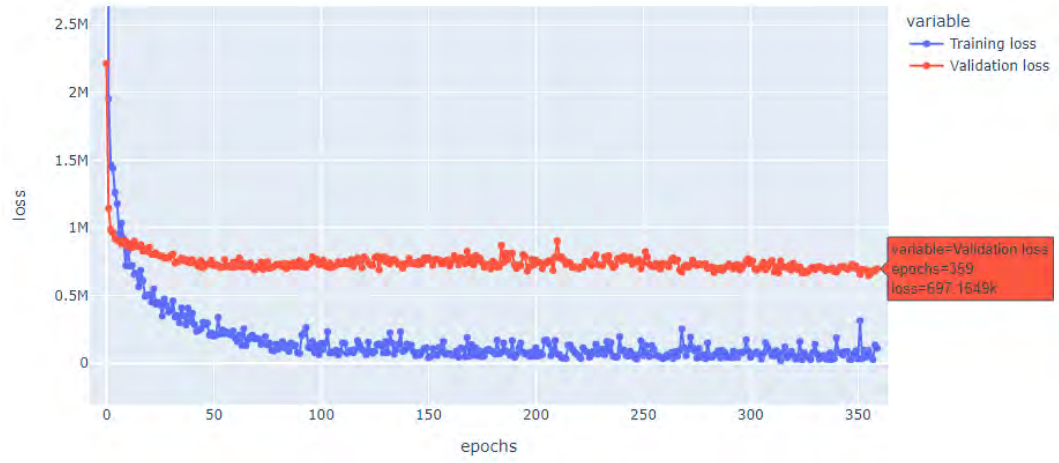
6.9. ábra. A *buildNN_RCNN* modell tovább tanítva. Az eredmény nem lett jobb.

6.3.3. Transfer learning

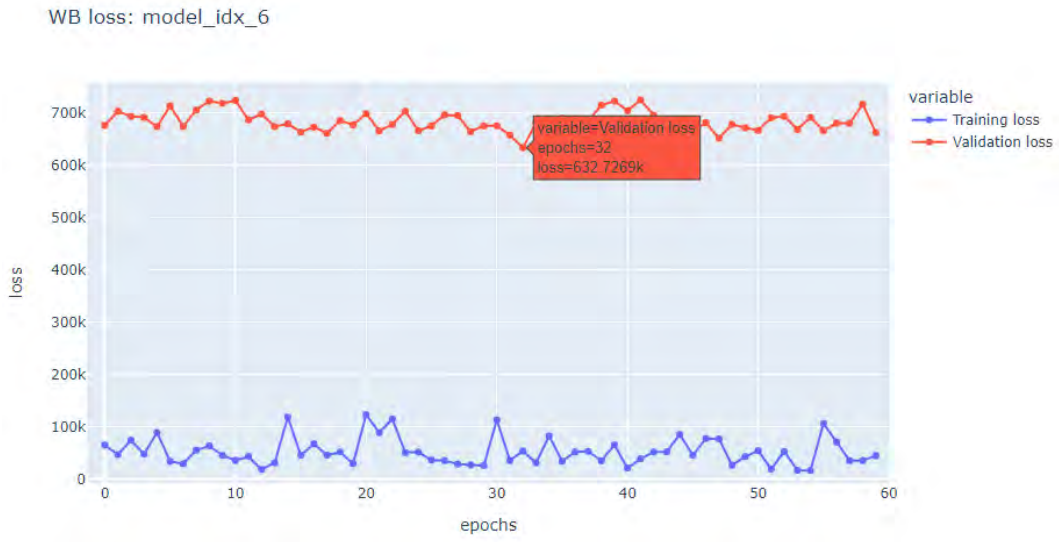
A transfer learning, mint módszer leírása korábban, a Transfer learning részben található. Ebben a részben a transfer learning alkalmazásáról esik szó.

6.3.3.1. InceptionV3 transfer

A következő modell a InceptionV3 háló alapjain lett tanítva, majd a korábbi ígéretes tapasztalatok miatt a *buildNN_init* teljesen csatolt rétegei (fully connected layers) kerültek az InceptionV3 tetejére, ezáltal transfer learninget megvalósítva.



6.10. ábra. Előtanulási fázis az InceptionV3 hálón



6.11. ábra. Utótanulási fázis az InceptionV3-on, a *buildNN_xs* teljesen csatolt (fully connected) rétegeivel

A 6.10 grafikonokon látható, hogy ez az alapmodell túl komplex a fehéregyensúly megtanulásához, a validációs hiba sajnos közelébe se ért a tanulási hibának, érdemes ezért egy egyszerűbb alapmodellt kiválasztani és azon tanítani a végleges hálót.

Még két másik, egyszerűbb hálón is kipróbáltam a *buildNN_xs* teljesen csatolt (fully connected) rétegeit az InceptionV3 után. A keras előre tanított modelljei közül a MobileNetV2-re és a DenseNet121-re esett a választásom, mert ezek a hálók kisebbek az InceptionV3 hálónál, ezekről összehasonlítás a 6.2 táblában látható:

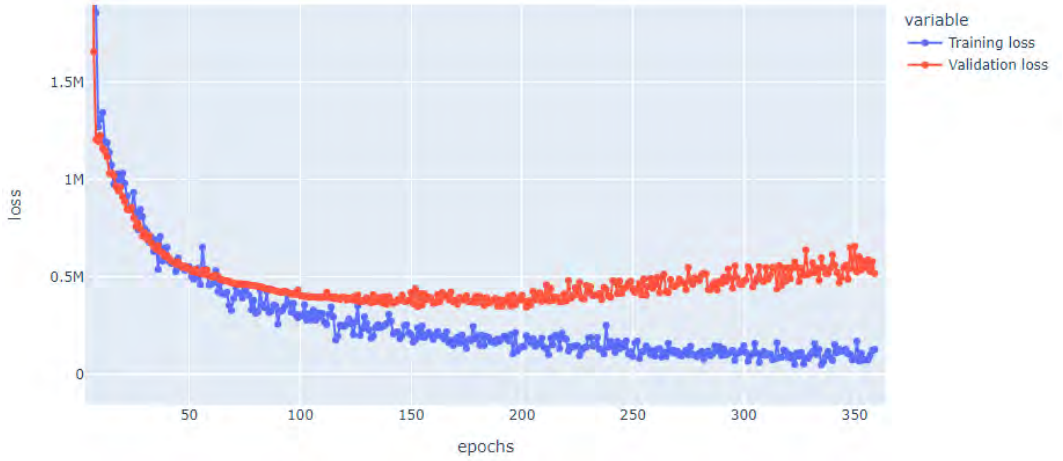
Model	Size (MB)	Top1-Acc	Top5-Acc	#params	depth	time/inference (ms, on CPU)	time/inference (ms, on GPU)
InceptionV3	22	77.9%	93.7%	23.9M	189	42.2	6.9
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4

6.1. táblázat. Választott alaphálók tulajdonságai

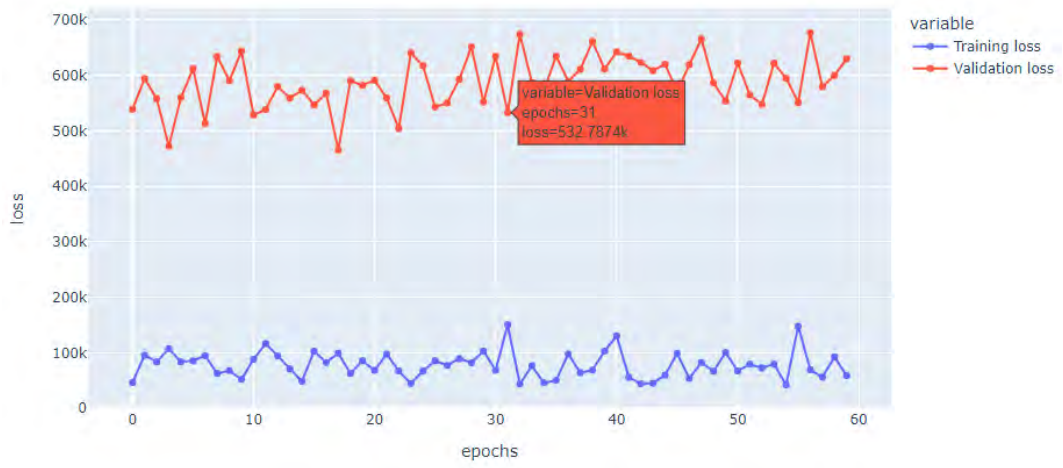
A táblázatban a paraméterek száma a teljes eredeti modellekre vonatkoznak. A korábbi, InceptionV3-ra épített háló csak 22 millió paraméterrel rendelkezett a teljesen csatolt rétegek neuronjainak alacsony száma miatt, míg itt ∼24 millió paramétert ír a táblázat. [26]

6.3.3.2. MobileNetV2 transfer

A MobileNetV2 alapjain bíztatóbb eredményeket vártam mint az InceptionV3 hálón, a kisebb hálóméret és paraméterszám miatt. Ennek ellenére sajnos még ez az modell is túl komplex, a tanulási görbéken a túltanulás (overfitting) figyelhető meg (a 150. epochtól), a korábban látott hálókhoz képest magas hiba mellett (∼400.000). Így a háló finomhangolásakor az utótanulási fázisban nem volt semmiféle elvárásom a javulásra, ez be is igazolódott.



6.12. ábra. A *modelNN_pretrained* háló előtanulási görbéje MobileNetV2 alapokon. Jól kivehető az overfitting



6.13. ábra. A *modelNN_pretrained* háló utótanulási görbéje a MobileNetV2 alapjain

A korábbi próbálkozások sikertelensége miatt arra jutottam, hogy a DenseNet121-re épített hálót már nem érdemes kipróbálni, várhatóan hasonló eredménnyel zárulna a

tanítása, mint a InceptionV3 és a MobileNetV2 hálóknak, hiszen több paramétere van és mélyebb is, mint a MobileNetV2.

6.3.4. Összefoglaló a modellek hibaértékeiről

Az eddigi modellek validációs halmazon mért hibáit az alábbi táblázat foglalja össze. Az AbsMean a hibák abszolút értékének átlagát, az Std a hibák abszolút értékének szórását jelölik.

	modelNN_init	modelNN_s	modelNN_xs	modelNN_RCNN	modelNN_pre
AbsMean	182,430	385,300	295,390	230,540	498,910
Std	154,720	298,770	269,110	144,840	371,270

6.14. ábra. Az eddig tárgyalt modellek hibastatisztikái

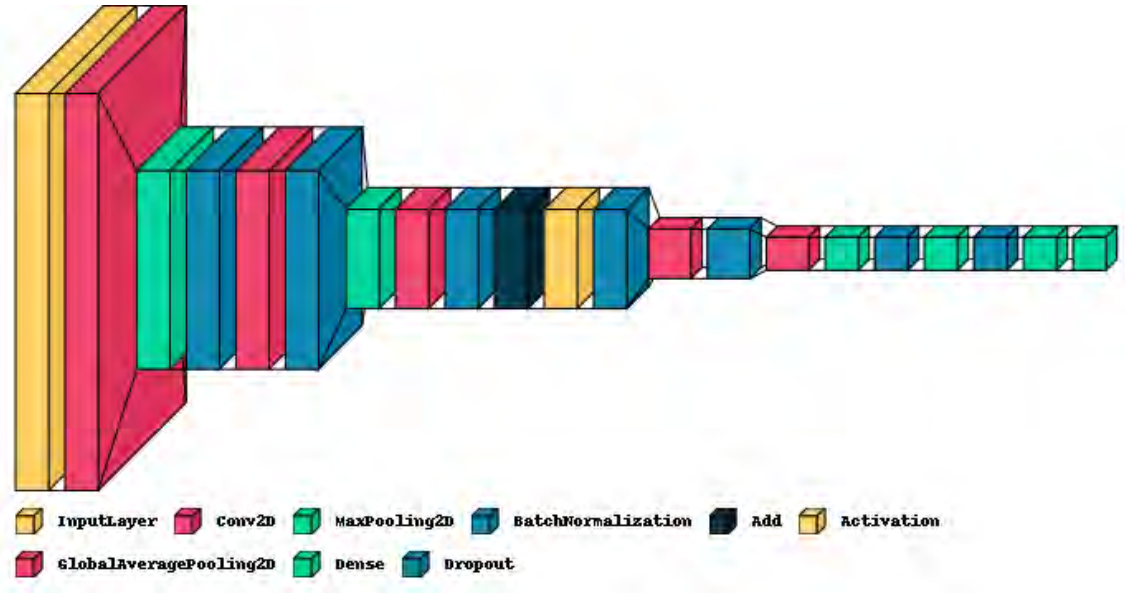
Az hibákat összevetve a *modelNN_init*, vagy a *modelNN_RCNN* architektúrkáját lehetne tovább fejleszteni, vagy kombinálni a két háló erősségeit. A korábbi grafikonokon ezeknél látszik, hogy elég robosztus modellek, illetve a táblázat alapján a hibák nagyságát és szórását tekintve is ezek a legígéretesebbek. Érdemes lehet még figyelembe venni a *modelNN_xs* hálót, ez adta a harmadik legjobb hibaértékeket, de egyszerűbb modell, mint a *modelNN_init*. A *modelNN_s* és a transfer learning láthatóan nem vált be, magasabb hibával dolgoznak.

	modelNN_init	modelNN_s	modelNN_xs	modelNN_RCNN	modelNN_pre
img_0	8,87	226,42	493,76	188,82	677,52
img_1	-97,14	472,56	1005,38	802,18	16,07
img_2	-199,58	36,73	147,6	65,86	561
img_3	-145,44	250,75	1054,76	484,58	1690,73
img_4	287,24	335,64	405,1	416,96	795,86
img_5	-128,02	200,51	361,08	309,84	-63,17
img_6	-177,48	12,1	545,01	169,65	23,67
img_7	30,57	188,44	483,75	146,55	-61,93
img_8	1004,82	746,38	925,99	867	671,36
img_9	-511,23	-137,48	-125,81	-295,22	1618,96
img_10	264,65	281,58	629,07	261,85	261,11
img_11	-252,9	-1339,88	795,62	-86,39	1316,48
img_12	-642	-137,84	-188,17	-53,7	-553,79
img_13	21,43	-195,16	538,13	95,74	667,99
img_14	18,6	312,05	499,78	208,9	908,32
img_15	51,81	434,11	1040,35	698,32	847,11
img_16	-62,76	59,18	215,23	119,25	-134,49
img_17	-46,97	468,82	1018	479,92	1114,66
img_18	253,12	377,89	380,15	347,8	911,08
img_19	-83,72	196,36	406,8	324,61	61,02
img_20	-90,89	89,3	552,61	174,18	-198,66
img_21	-101,54	153,49	504,35	136,12	456
img_22	866,67	822,18	880,17	857,33	620,98
img_23	-455,32	-98,64	-163,28	-222,87	-217,61
img_24	411,57	358,92	653,35	308,05	1717,1
img_25	-123,53	-1008,44	546,34	-122,07	1185,22
img_26	-347,19	-53,27	-37,62	-7,42	113,3
img_27	368,47	-69,94	427,22	126,79	797,57
AbsMean	182,430	385,300	295,390	230,540	498,910
Std	154,720	298,770	269,110	144,840	371,270

6.15. ábra. Hálók hibáinak részletes összehasonlítása a validációs halmazon

6.3.5. Még egy visszacsatolt konvolúciós háló

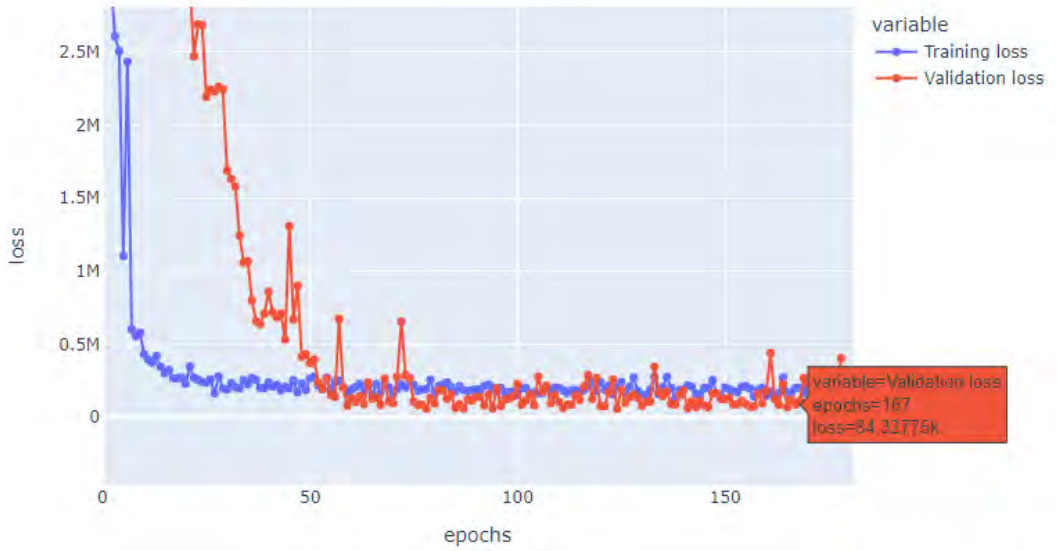
A fenti statisztika alapján érdemesnek találtam a *modelNN_init*, és a *modelNN_RCNN* hálók tulajdonságait egyesíteni, hiszen ezek adták a legjobb hibaértékeket. Az új háló a *modelNN_irv2* lett.



6.16. ábra. A *modelNN_irv2* architektúrája

Az ábrán sajnos megtévesztő lehet a jelmagyarázat az azonos színű rétegek különböző rétegeire, az egyértelmű felépítés az alábbi:

Input → Conv2D → MaxPooling2D → BatchNormalization → Conv2D → BatchNormalization → MaxPooling2D → Conv2D → BatchNormalization → Add → Activation → BatchNormalization → Conv2D → BatchNormalization → GlobalAveragePooling2D → Dense → Dropout → Dense → Dropout → Dense → Dense



6.17. ábra. A *modelNN_irv2* modell tanulási görbéje

Jelentős minőségi javulást sajnos nem lehet észrevenni, az elődmodellekhez hasonlóan 80-100 ezres hibák fordulnak elő. Ezen kívül ahogy korábban is meg lehetett figyelni, elég

nagy ugrások vannak a hibaértékekben. Ez valószínűleg a kevés és nem elég reprezentatív adatnak köszönhető, az igazán jó működéshez ugyanis milliós nagyságrendbe kellene elérhető és jó tanulási képadatoknak lennie a prediktálni kívánt tulajdonságokkal együtt, sajnos ilyen adathalmaz jelenleg nincs. Ezzel a problémával ugyanakkor a többi modellnek is meg kell küzdenie.

6.3.6. Összehasonlítás különböző adathalmazokon

A dolgozat ezen részében az eddigi modellek hibáit hasonlítom össze a 3 különböző adathalmazon. A negyedik táblázatrész a Mixed adathalmaz négyszeres augmentálásával jött létre. Ez magában hordozza a túltanulás veszélyét, de early stopping segítségével kivédhető.

	Errors measured on Konstanz dataset					
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_pre**	**modelNN_irv2**
AbsMean	224,11	312,71	357,89	167,47	601,29	177,27
Std	159,72	220,07	239,98	165,96	389,13	172,04
	Errors measured on Szakestély dataset					
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_pre**	**modelNN_irv2**
AbsMean	252,98	490,43	339,76	297,60	530,00	294,30
Std	183,65	307,98	247,25	218,13	333,81	240,04
	Errors measured on Mixed dataset					
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_pre**	**modelNN_irv2**
AbsMean	226,75	362,62	519,86	262,53	607,77	258,93
Std	211,06	270,48	278,77	232,60	513,95	174,04
	Mixed Superaugmented (x4)					
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_pre**	**modelNN_irv2**
AbsMean	202,08	308,46	435,58	297,91	909,00	220,20
Std	166,05	213,52	225,09	222,86	536,10	191,98

6.18. ábra. A hálók hibáinak összehasonlítása

Zölddel az legjobb, míg sárgás színnel a második, vagy holtversenyben második modellek lettek jelölve.

Ezek alapján a legjobb választásnak a *modelNN_init* tűnik, ugyanakkor nincsenek jelentős különbségek *modelNN_irv2*-lel.

6.3.7. Különböző modellparaméterezések

Általánoságban a MSE (mean square error) jó választás regressziós problémákhoz, ugyanakkor érdemes pár másik hibafüggvényt is megnézni, például a MAE-t (mean absolute error) vagy a Huber-losst. Optimalizálók területén hasonló a helyzet, az utóbbi időben az Adam optimalizálót használják a sok modellhez, egy jól bevált megoldás. Ennek ellenére az Adagrad és az RMSProp optimalizálókkal is kísérleteztem. Érdemes lehet még különböző batch méreteket megpróbálni, nagyobb batch mérettel elméletileg pontosabb a háló konvergenciája egy lépésben, mivel több minta hibájából számítódik a gradiens. Ezeket a változtatásokat a *modelNN_init* és a *modelNN_irv2* modelleken próbáltam ki a Mixed adathalmazt használva.

6.3.7.1. Hibafüggvények

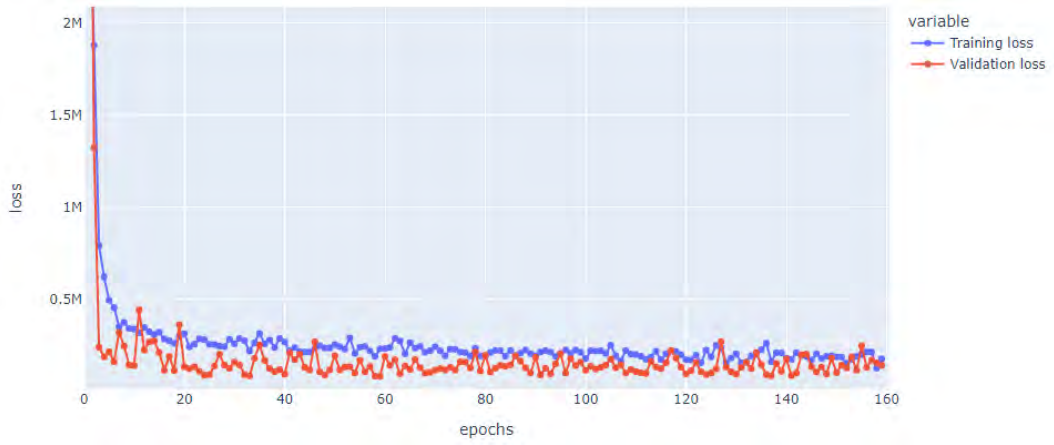
Loss type		modelNN_init	modelNN_irv2
MAE	AbsMean	261,90	210,16
	Std	182,70	189,65
Huber	AbsMean	247,18	196,36
	Std	227,49	169,27
MSE	AbsMean	226,75	258,93
	Std	211,06	174,04

6.19. ábra. A különböző hibafüggvények hálókra gyakorolt hatása

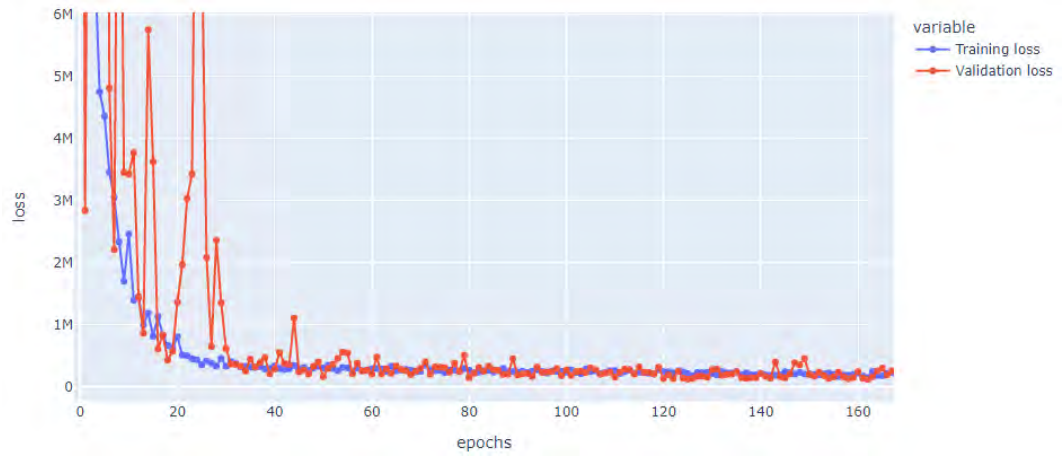
A méréseik alapján úgy látszik, hogy nem teljesen mindegy, hogy milyen hibafüggvények alapján történik a hálóparaméterek frissítése, a Huber loss igen jótékony hatással volt a *modelNN_irv2* modellre, de a MAE (mean absolute error) is javított a teljesítményen. Emiatt a továbbiakban a *modelNN_irv2* hibafüggvénye Huber loss lesz, a *modelNN_init* hibafüggvénye marad MSE (mean square error).

6.3.7.2. Optimalizálók

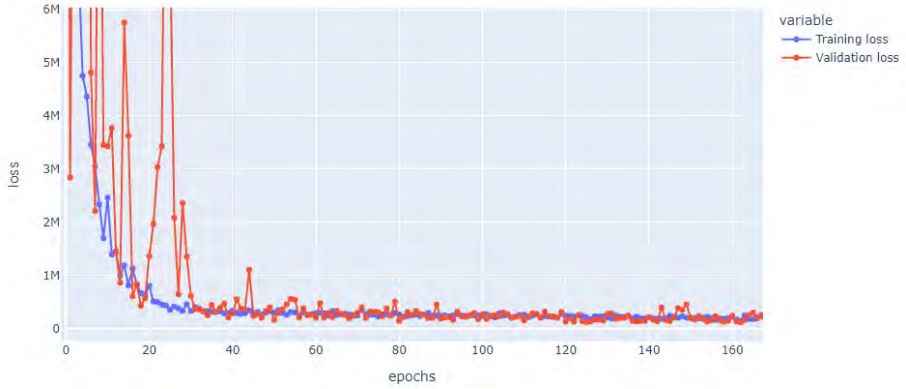
Az optimalizálók a lépésenkénti gradiensfrissítés mértékét befolyásolják, így a konvergencia sebességét is, ezért itt a tanulási görbéket érdemes kiértékelni. Az lent található tanulási görbék a *modelNN_init* és a *modelNN_irv2* hálók egy tanítását mutatják be Adam, RMSProp és az Adadelta optimalizálókkal.



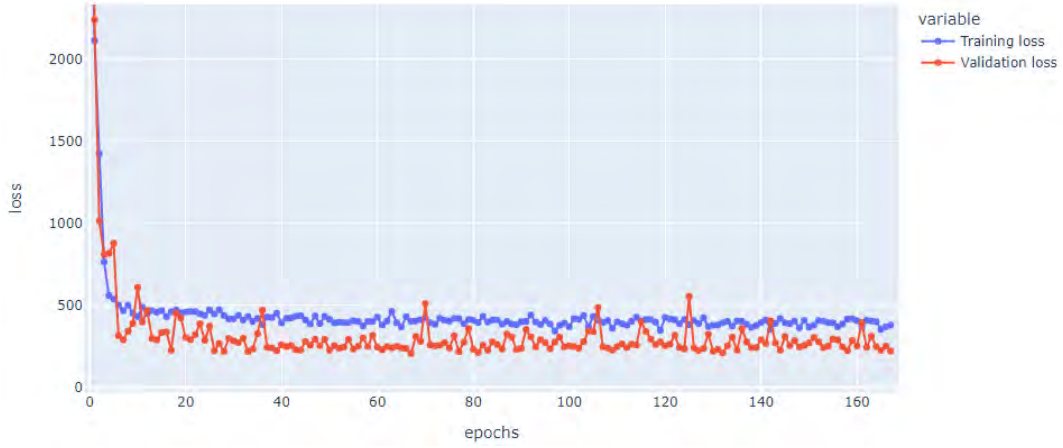
6.20. ábra. *modelNN_init* háló tanulási görbéje, Adam optimizer



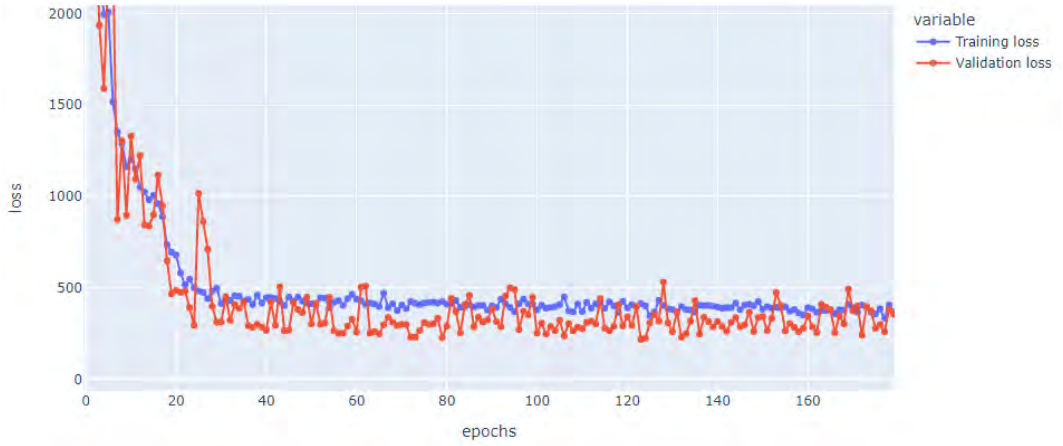
6.21. ábra. *modelNN_init* háló tanulási görbéje, RMSProp optimizer



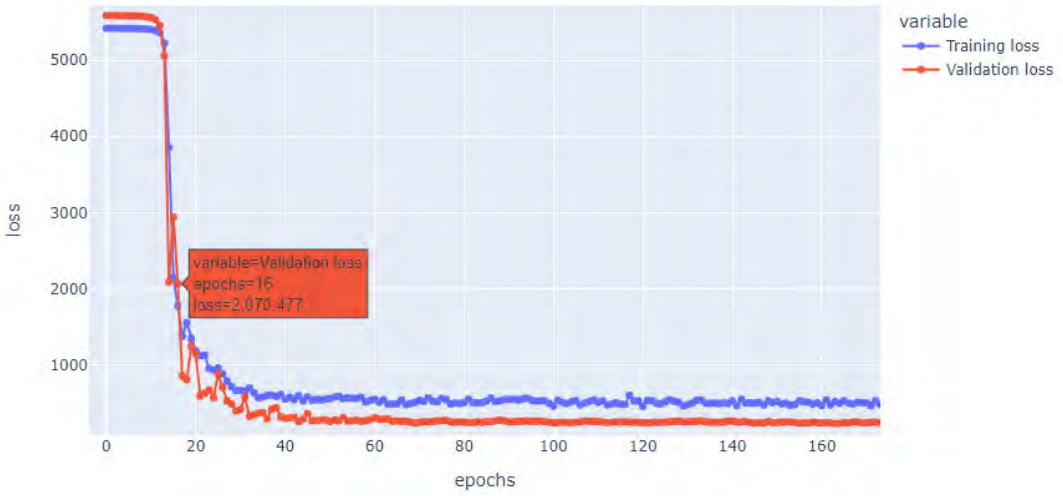
6.22. ábra. *modelNN_init* háló tanulási görbéje, Adadelta optimizer



6.23. ábra. *modelNN_irv2* háló tanulási görbéje, Adam optimizer



6.24. ábra. *modelNN_irv2* háló tanulási görbéje, RMSProp optimizer



6.25. ábra. *modelNN_irv2* háló tanulási görbéje, Adadelta optimizer

Összehasonlítva a fentebbi tanulási görbéket, észrevehetjük, hogy az Adam volt mind közül a leghatékonyabb. A tanítás korai szakaszaiban Adam-et használva kevésbé voltak ugráló validációs hibák, mint az RMSProp optimalizálót használva. A tanulás egyenletességét nézve az Adadelta a legjobb (ábrák a függelékben). Az Adadelta kezdetben nem tesz érdemi optimalizációs lépéseket, a hibaértékek sokáig azonos szinten maradnak, majd utána kezd el csak el csökkenni. Ezt a másik két optimalizáló hamarabb eléri a tanulás kezdetén tett drasztikusabb hibacsökkentéssel.

A következő táblázat alapján ugyan annyi epoch alatt az Adam-mel tanuló hálók jobban teljesítettek, mint a másik két optimalizálóval tanuló változatok, valamint a *modelNN_irv2* háló előnye is egyérelművé válik.

Optimizer type		modelNN_init	modelNN_irv2
Adam	AbsMean	226,75	196,36
	Std	211,06	169,27
RMSProp	AbsMean	259,40	203,96
	Std	203,73	160,83
Adadelta	AbsMean	326,17	212,67
	Std	224,06	180,51

6.26. ábra. Optimalizálók összehasonlítása a validációs hibákat tekintve. A *modelNN_irv2* mindenhol megveri a *modelNN_init*-et.

6.3.7.3. Batch méretek

A batch méret azt határozza meg, hogy hány mintát vesz a háló egyszerre, amik alapján hibát számol és súlyt frissít. Például a 4-es batch méret azt jelenti, hogy egy tanulási cikluson (epoch) és annak egy lépésén belül 4 mintát küld előre a háló, veszi ezek hibáját és ez alapján frissít a hálón, azaz minden 4 mintára egy súlyfrissítés jut.

A tanítás során a 1, 4, 8, 16 és 32-es batch méreteket találtam próbára érdemesnek. Itt a validációs halmazon mért hibák számtani közepe és szórása ad egy képet a batch méret jóságáról. Úgy látszik, hogy a Mixed adathalmazon tanítva a 4-es batch méret tűnik a legjobbnak, ugyanakkor a 8-as batch méret is még elfogadhatóan jónak tűnik a *modelNN_irv2* esetén. Az egyes batch méretnél nem meglepő az ennyire magas hibaérték, hiszen itt minden egyes minta után súlyfrissítésre kerül sor, így a tanulás nagyon instabil, hiszen mindig egy aktuális minta húzza el a súlyokat a neki megfelelő irányba, nem egy közös minimum felé tart a háló.

Batch size		modelNN_init	modelNN_irv2
1	AbsMean	435,96	358,05
	Std	293,67	327,90
4	AbsMean	226,75	196,36
	Std	211,06	169,27
8	AbsMean	280,39	202,75
	Std	233,78	227,30
16	AbsMean	240,00	221,67
	Std	202,26	203,21
32	AbsMean	277,60	232,84
	Std	227,26	168,96

6.27. ábra. A *modelNN_init* és a *modelNN_irv2* validációs halmazon mért hibái különböző batch méretek mellett (zöld: legjobb érték, sárgás: második legjobb érték).

6.3.8. Végleges hálók a tulajdonságok prediktálására

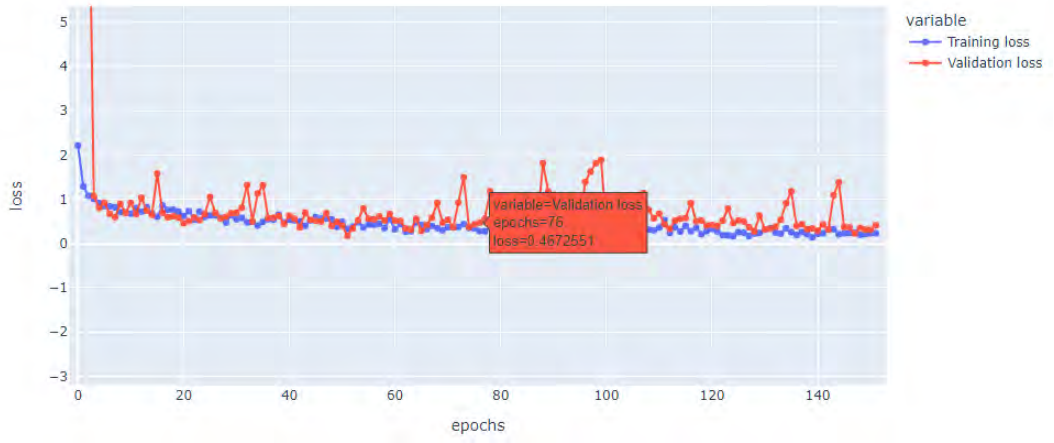
A korábbi részekben csak a fehéregyensúly prediktálása volt cél, e tulajdonság alapján lettek a hálók kiértékelve, viszont a többi tulajdonságot is jó lenne prediktálni.

A korábbi tapasztalatok alapján a *modelNN_irv2* háló jó lesz a fehéregyensúly prediktálására.

Az árnyalatok (lila-zöld egyensúly, vagy tint) prediktálása nem volt egyszerű feladat, az első tanítások során nagyon instabil eredmények születtek, amiket nem tudott a háló jól lefedni, illetve túltanult a modell, a validációs halmazon nem tudott jó eredményt elérni. Valószínűleg a Szakest és ezáltal a Mixed adathalmaz kiegyensúlyozatlansága állt a háttérben (ezek 5-6 közötti értéket adtak az AbsMean-re, míg a szórásra 4-5 közötti volt az érték), mert a Konstanz adathalmazon értelmes eredmények születtek és a modell se tanult túl, illetve az esetleges túltanulást az early stopping akadályozza.

	Errors measured on Konstanz dataset (tint)				
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_irv2**
AbsMean	0,62	0,63	0,76	0,89	0,47
Std	0,66	0,97	0,65	0,83	0,41

6.28. ábra. A különböző modellek hibastatisztikái árnyalatot prediktálva (tint)



6.29. ábra. A *modelNN_irv2* háló tanulási görbéje az árnyalatra (tint)

A világosság prediktálásához elég jó eredményt ért el a *modelNN_irv2* háló a 0,18-as átlaghibával. A világosság értékei a [-5;+5] tartományba esnek (a Lightroomban ekkora tartományban értelmezett ez a paraméter). Ilyen értékek mellett a produkált hiba elfogadható, valamint több adattal várhatóan tovább leszorítható, csakúgy, mint a többi tulajdonság hibája.

	Errors measured on Mixed dataset (brightness/exposure)				
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_irv2**
AbsMean	0,23	0,22	0,20	0,21	0,18
Std	0,24	0,21	0,22	0,24	0,21

6.30. ábra. A hálók hibastatisztikája a világosságot prediktálva

A kontrasztot legjobban prediktáló háló meglepő módon a *modelNN_init* lett, a többi háló elmarad ettől, ahogy ez az alábbi táblázatban látható. A kontraszt elméleti értékkészlete a [-100;+100] intervallumba esik, de nem szoktak nagyon a [0;+10] tartományon kívül értékek előfordulni.

	Errors measured on Mixed dataset (contrast)				
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_irv2**
AbsMean	0,20	0,61	0,57	0,46	0,28
Std	0,21	1,07	0,90	0,83	0,34

6.31. ábra. A hálók hibaértékei kontraszt prediktálásánál

A vibrálást ismét a *modelNN_irv2* hálónak sikerült a legjobban eltalálnia, így erre a tulajdonságra is érdemes ezt a hálót alkalmazni. A vibrálás elméleti értékkészlete a [-100;+100] intervallum, de a gyakorlatban nem igazán fordulnak elő az [-5;+20] intervallumon kívüli értékek.

	Errors measured on Mixed dataset (vibrance)				
Models	**modelNN_init**	**modelNN_s**	**modelNN_xs**	**modelNN_RCNN**	**modelNN_irv2**
AbsMean	0,45	0,81	0,61	0,45	0,23
Std	0,95	1,09	1,37	0,79	0,52

6.32. ábra. A modellek hibái vibrálás esetén

Az 5 tulajdonság prediktálásához tehát az alábbi hálók adnak a megadott körülmények mellett elfogadható értékeket:

Feature/property	**NN model**	**Optimizer**	**Loss**	**Batch size**
White balance (WB):	*modelNN_irv2*	Adam	Huber	4
Tint:	*modelNN_irv2*	Adam	MAE	4
Exposure:	*modelNN_irv2*	Adam	MSE	4
Contrast:	*modelNN_init*	Adam	Huber	4
Vibrance:	*modelNN_irv2*	Adam	Huber	4

6.33. ábra. Választott modellek és főbb paramétereik a tulajdonságok prediktálásához

6.4. Betanult modellek mentése, betöltése

A betanított modellek HDF5 fájlokként (.h5 kiterjesztés), opcionálisan dátummal ellátva mentésre kerülnek Google Drive-ra, ezt végzi a *save_models* függvény. A *tf.Keras.Model* osztálynak a modellek mentéséhez és a betöltéshez beépített függvénye van, később lokálisan a modellek visszatöltéséhez a *load_models* függvény szükséges. [46] [19]

6.5. Futtatás előtti konfigurációk

A futtatási környezetben a python 3.9-es verziója, illetve az alkalmazáshoz szükséges csomagok szükségesek, ezeket a pip csomagkezelővel lehet könnyedén letölteni.

Ahhoz, hogy az alkalmazás meg tudja nyitni az Adobe Lightroom-ot a predikció végén, szükséges ez az alkalmazás is. Be kell állítani a Lightroom Autoimport funkcióját is egy tetszőleges, még üres mappára. Ez ahhoz szükséges, hogy a Lightroom automatikusan fel tudja olvasni azokat a képeket, amikhez a korrekciós fájlok elkészültek. Érdemes az *LR* környezeti változóban eltárolni a Lightroom.exe-re mutató útvonalat, az alkalmazás először megpróbálja ezen keresztül a Lightroom-ot indítani. Ez az elérési út általában így néz ki: C:\Program Files\Adobe\Adobe Lightroom Classic\Lightroom.exe‘. A környezeti változó beállítása után előfordulhat, hogy újra kell indítani a számítógépet.

Végül le kell tölteni a betanított modelleket és egy új mappába kell őket másolni, ahol csak a használni kívánt modellek vannak.

6.6. Jellemzők prediktálása

A projekt ezen része az, ami a felhasználók számára készült, a *predictor.py* egy CLI alkalmazás, ami az alábbi paramétereket várja:

6.6.1. Kötelező paraméter(ek)

`raw_source_dir`: A prediktálni kívánt nyersképek elérési útja.

6.6.2. Opcionális paraméterek

`-lr_target_dir`: Az a mappa, amire a Lightroom Autoimport be van állítva, alapbeállítása: `./resources/auto_imp_dir`

`-path_to_models`: A modellek elérési útja, alapbeállításban `./models`

`-open_up_LR`: True vagy False értéket vehet fel, a prediktálás után a Lightroom indítását kapcsolja be (True), vagy ki (False). Alapbeállítása True.

`-lr_executable_path`: A Lightroom.exe elérési útja, ha nincs az LR környezeti változó beállítva, alapbeállítása C:\Program Files\Adobe\Adobe Lightroom Classic\Lightroom.exe‘.

`-create_copy`: True vagy False értéket vehet fel, a prediktálás után a fájlok áthelyezését vagy másolását állítja. Ha True, akkor lemásolja a `raw_source_dir`-ben lévő fájlokat az `lr_target_dir`-be, egyébként áthelyezi őket. Alapbeállítása False, azaz áthelyezés történik.

`-verbose`: A kimeneti üzenetek részletességét szabályozza.

6.1. lista. A *predictor.py* alkalmazás segítsége.

```
(python) PS C:\Users\Vince\Desktop\7_sem\bscThesis\scripts_new> python .\predictor.py .\resources\
    konstanz_test_set\ -h
usage: predictor.py [-h] [--lr_target_dir LR_TARGET_DIR] [-m PATH_TO_MODELS] [-o {True,False}] [-LRe
    LR_EXECUTABLE_PATH] [-c {True,False}] [-v] raw_source_dir

This program helps speeding up retouch workflows

positional arguments:
  raw_source_dir        path to the raw image source directory

optional arguments:
  -h, --help            show this help message and exit
  --lr_target_dir LR_TARGET_DIR
                        target dir for Lightroom open, here will the predictions and the original
    photos land
  -m PATH_TO_MODELS, --path_to_models PATH_TO_MODELS
                        Specify the path to the models, which are required to predict retouch values.
     Default is ./models
  -o {True,False}, --open_up_LR {True,False}
                        Switches ON/OFF auto-Lightroom start
  -LRe LR_EXECUTABLE_PATH, --lr_executable_path LR_EXECUTABLE_PATH
                        Specify where to find Lightroom executable, default: C:\Program Files\Adobe\
    Adobe Lightroom Classic\Lightroom.exe
  -c {True,False}, --create_copy {True,False}
                        Leave raw files where they were or bring them under Lightroom, default is
    False
  -v, --verbose         Verbosity switch
(python) PS C:\Users\Vince\Desktop\7_sem\bscThesis\scripts_new>
```

6.6.3. Az alkalmazás futása

Az alkalmazás először betölti a modelleket a *load_models* függvénnyel, ami modellek listáját adja meg.

Ez után a *prepare_to_prediction* függvény beolvassa a képeket és eltárolja a hálónak megfelelő formátumban, azaz minden képet egy [133x200x3] méretű mátrixként, majd visszaadja ezeket egy listában, a képek neveivel együtt.

A következő lépésben minden kép végigmegy a neurális hálókon (*predict_on_models függvény*) és létrejön egy lista, amiben minden képhez egy allista tartalmazza a korrekciós jellemzőket.

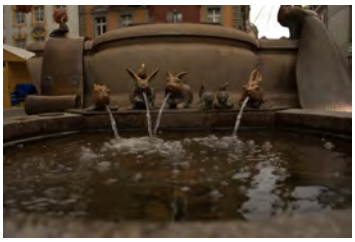
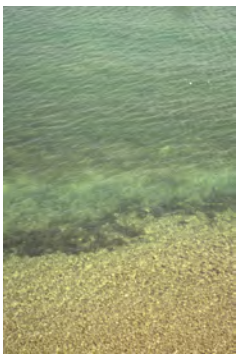
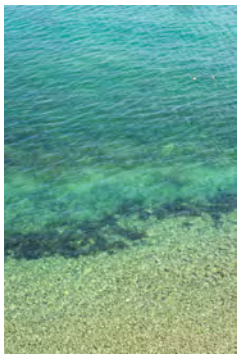
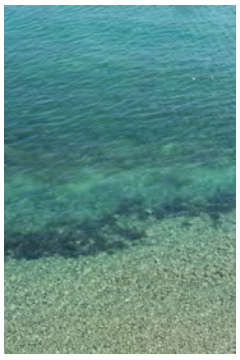
Ezt a korrekciós listát és a képek neveit kapja meg a *generate_batch_xmps* függvény, ami minden képhez legyártja a *generate_xmp_result* hívás segítségével a korrekciós fájlokat.

Az XMP fájlformátumhoz sajnos nincs igazán kézreeső és jól használható szerializáló könyvtár, ami lenne, az csak Linux rendszereken fut. A Lightroom viszont csak Windows rendszeren fut, ezért az .xmp részleteket, amely nagyjából 20 sorból áll minden esetben, egyszerű stringként kerül egy .xmp kiterjesztésű fájlba. A gyakorlatban a Lightroom ezt is szépen, probléma nélkül feldolgozza és alkalmazni tudja a leírt módosításokat.

Végül, ha ez szükséges, akkor a program átmásolja a képeket a Lightroom Autoimportja által megfigyelt könyvtárba. Ha be volt kapcsolva a Lightroom automatikus megnyitása, akkor meghívja a Lightroom.exe-t, ami indulás után automatikusan importálja a nyersképeket és a hozzájuk készített .xmp fájlokat, így előállnak az előre korrigált képek.

A háló által prediktált eredmények ekkor rögtön láthatók, ha valamit nem sikerült jól eltalálnia a hálónak, akkor a Lightroomban ez még kézzel javítható. Erre valószínűleg szükség is lesz, hiszen a gépi tanulás nem tud minden bemenetre jó megoldást adni, csak egy általa optimálisnak vélt megoldást. A tökéletes eredménynek ezen kívül az adathalmazok végessége és kis mérete is korlátot szab. [9] [49]

Zárásként pár képhármas található alább, ahol a nyerskép, az alapigazságnak elismert retusált változat, illetve a hálók által prediktált változat látható ugyan azon képből. Ugyan a neurális hálók által retusált képek még nincsenek a kézzel retusáltak közelében (ezeken más tulajdonságok is át lettek állítva), de mindenképp előremutató, amit eddig a hálók elértek.

Raw image	**Retouched etalon**	**Neural net retouched**
		
		
		
		
		

6.2. táblázat. Eredmények

7. fejezet

Munka értékelése, továbbfejlesztési lehetőségek

Szakdolgozat során számos új problémával kellett szembesülni és megoldani, amik korábbi tanulmányaim alatt, az elméleti alapok elsajátításakor nem merültek fel. Ilyen volt többek között az adatok előfeldolgozása és a kis adatmennyiség. Sajnos talán ezek miatt nem lettek elég pontosak a predikciók, így jelenleg még mindig pontosabb a választott 5 tulajdonságra az emberi retusálás, ugyanakkor ez csak adatmennyiség kérdése elviekben. A fentebbi fejezetekben tárgyalt eredmények a dolgozat írásakor rendelkezésre álló feltételek mellett érvényesek, ezek közé tartozik az adathalmaz minősége és mérete is.

A dolgozat írása és a modellválasztás közben számos továbbfejlesztési és kiegészítési ötletem támadt, ezek közül alább található pár.

Az adatmennyiség volt az egyik legnagyobb ellenfél. Erre lehetne a *process_data.py* alapján egy kliensalkalmazást írni, ami az adatgyűjtéssel foglalkozik a felhasználók beleegyezésével és hozzájárulásával. Ez egy központi repository-ba töltené fel a már feldolgozott adatokat. A folyton frissülő adatok alapján ütemezetten és folyamatosan lehetne DevOps környezet beállításával a modelleket tanítani és frissíteni a klienseknél.

Ha már jóval nagyobb adathalmaz áll rendelkezésre, akkor érdemes lehet a transfer learning lehetőségét újra elővenni, illetve a tárgyalt modelleket újraértékelni. Mivel ez rengeteg idő manuálisan végrehajtva, ezért egy DevOps környezettel és automatikus hiperparaméter (modellparaméter) optimalizálással ismét gyorsítani lehetne a folyamaton. Ehhez azonban nagyobb és viszonylag korlátlan számítási kapacitás szükséges.

Érdemes lehet a hálókat nem külön-külön minden egyes képtulajdonságra tanítani, hanem akár egy alap konvolúciós hálóból leágazva több teljesen kapcsolt (fully connected) réteget létrehozni, amik az egyes tulajdonságokat prediktálják. Illetve a hasonló fontosságú és értéktartományú paramétereket akár egy háló is prediktálhatná, ezzel tárterületet megspórolva. Ez jelenleg azért nem lett így megoldva, mert a fejlesztés korai szakaszában kiderült, hogy a fehéregyensúly értéktartománya nagyon eltér és ezáltal elhúzza a hálót egy irányba. Emiatt a többi tulajdonság prediktálása rossz eredményeket ad, ezért minden tulajdonság külön hálót kapott.

Az jelenleg parancssoros *predictor.py* alkalmazás felé lehetne írni egy grafikus alkalmazást, amivel a kevésbé jártas felhasználók számára egyszerűbb lenne az alkalmazás használata.

Összefoglalva a kitűzött célok megvalósultak, de számos finomítási, pontosítási és továbbfejlesztési lehetőség maradt nyitva, amikkel jó lenne a későbbiekben még tovább foglalkozni.

Köszönetnyilvánítás

Szeretném megköszönni konzulensemnek, Dr. Ekler Péternek a félévben nyújtott segítséget.

Köszönettel tartozom szüleimnek, akik a kilátástalannak tűnő helyzetekben is támogattak, érdeklődve és türelmesen végighallgatták az éppen felmerülő problémákat vagy megoldási ötleteim, amiken dolgoztam a szakdolgozat alatt.

Szeretném megköszönni két Simonyis ismerősömnek a hasznos és tartalmas beszélgetéseket a neurális hálókkal kapcsolatban, ezek igen sokat segítettek, valamint szobatársaimnak a támogatást.

Végül szeretném megköszönni mindazoknak, akiknek szerepe volt abban, hogy a német nyelvű képzés végén egy évet tölthettem a Karlsruher Institute für Technologie-n (KIT), ahol megismerkedtem a gépi tanulással és a neurális hálókkal és megszerettem ezt a témát.

Irodalomjegyzék

[1] Adadelta — ML Glossary, 2017. URL https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adadelta. [Online; accessed 7-December-2022].

[2] Adagrad — ML Glossary, 2017. URL https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adagrad. [Online; accessed 7-December-2022].

[3] Adam — ML Glossary, 2017. URL https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adam. [Online; accessed 7-December-2022].

[4] Adobe: XMP Specification. URL https://www.adobe.com/devnet/xmp.html. [Online; accessed 7-December-2022].

[5] Andrea Apicella – Francesco Donnarumma – Francesco Isgrò – Roberto Prevete: A survey on modern trainable activation functions. *Neural Networks*, 138. évf. (2021), 14–32. p. ISSN 0893-6080. URL https://www.sciencedirect.com/science/article/pii/S0893608021000344. [Online; accessed 7-December-2022].

[6] Artifical intelligence — Wikipedia, The Free Encyclopedia, 2022.12. URL https://en.wikipedia.org/wiki/Artificial_intelligence. [Online; accessed 7-December-2022].

[7] Artifical neural network — Wikipedia, The Free Encyclopedia, 2022.12. URL https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg. [Online; accessed 7-December-2022].

[8] Batch Normalization — Wikipedia, The Free Encyclopedia, 2022.10.03. URL https://en.wikipedia.org/w/index.php?title=Batch_normalization&oldid=1113831713. [Online; accessed 7-December-2022].

[9] Jason Brownlee: How To Improve Deep Learning Performance (in Deep Learning Performance), 2019.08. URL https://machinelearningmastery.com/improve-deep-learning-performance/. [Online; accessed 7-December-2022].

[10] Jason Brownlee: How to use Learning Curves to Diagnose Machine Learning Model Performance (in Deep Learning Performance), 2019.08. URL https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/. [Online; accessed 7-December-2022].

[11] Claude Shannon — Wikipedia, The Free Encyclopedia, 2022.11.13. URL https://en.wikipedia.org/w/index.php?title=Claude_Shannon&oldid=1121721760. [Online; accessed 7-December-2022].

[12] Dataset — HuggingFace. URL https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset. [Online; accessed 7-December-2022].

[13] Budapesti Műszaki és Gazdaságtudományi Egyetem. Dániel, Hadházi Méréstechnika és Információs Rendszerek Tanszék (MIT): Mély konvolúciós neurális hálózatok, 2019.
URL http://home.mit.bme.hu/~hadhazi/Oktatas/NN19/Mely_CNN.pdf. [Online; accessed 7-December-2022].

[14] FileInfo.com: ARW File Extension. URL https://fileinfo.com/extension/arw. [Online; accessed 7-December-2022].

[15] FileInfo.com: CR2 File Extension. URL https://fileinfo.com/extension/cr2. [Online; accessed 7-December-2022].

[16] FileInfo.com: XMP File Extension. URL https://fileinfo.com/extension/xmp. [Online; accessed 7-December-2022].

[17] María García-Ordás – José Benítez-Andrades – Isaías García – Carmen Benavides – Hector Alaiz Moreton: Detecting respiratory pathologies using convolutional neural networks and variational autoencoders for unbalancing data. *Sensors*, 20. évf. (2020. 02). [Online; accessed 7-December-2022].

[18] Gerhard, Neumann — Autonome Lernende Roboter (ALR), Institut für Anthrophomatik und Robotik, KIT: Organization + Introduction, 2021.10. URL https://ilias.studium.kit.edu/goto.php?target=file_1312284_download&client_id=produktiv. [Online; accessed 7-December-2022].

[19] HDF5 files in Python — GeekforGeeks. URL https://www.geeksforgeeks.org/hdf5-files-in-python/. [Online; accessed 7-December-2022].

[20] Johann Huber: Batch normalization in 3 levels of understanding., 2020.11.06. URL https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338. [Online; accessed 7-December-2022].

[21] Iconiq Inc.: Introduction to Convolutional Neural Networks Architecture, 2022.09.15. URL https://www.projectpro.io/article/introduction-to-convolutional-neural-networks-algorithm-architecture/560. [Online; accessed 7-December-2022].

[22] Inception-v3. — Paperswithcode.
URL https://paperswithcode.com/method/inception-v3. [Online; accessed 7-December-2022].

[23] InceptionResNetV2 — Keras API reference, Keras Applications.
URL https://keras.io/api/applications/inceptionresnetv2/. [Online; accessed 7-December-2022].

[24] John McCarthy — Wikipedia, The Free Encyclopedia, 2022.10.13. URL https://en.wikipedia.org/w/index.php?title=Marvin_Minsky&oldid=1121915033. [Online; accessed 7-December-2022].

[25] Aleksey Bilogur Kaggle: Notes on residual connections, 2019. URL https://www.kaggle.com/code/residentmario/notes-on-residual-connections. [Online; accessed 7-December-2022].

[26] Keras Applications — Keras API reference.
URL https://keras.io/api/applications/. [Online; accessed 7-December-2022].

[27] Lectures of Deep Learning Neural Networks. KIT.
URL https://isl.anthropomatik.kit.edu/english/3058_3069.php. [Online; accessed 7-December-2022].

[28] Marvin Minsky — Wikipedia, The Free Encyclopedia, 2022.11.14. URL https://en.wikipedia.org/w/index.php?title=Marvin_Minsky&oldid=1121915033. [Online; accessed 7-December-2022].

[29] Max and average pooling picture, 2022.08. URL https://tex.stackexchange.com/questions/654546/tables-side-by-side-with-big-curly-brackets. [Online; accessed 7-December-2022].

[30] MobileNetV2 — Paperswithcode.
URL https://paperswithcode.com/method/mobilenetv2. [Online; accessed 7-December-2022].

[31] Momentum — ML Glossary, 2017. URL https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#momentum. [Online; accessed 7-December-2022].

[32] Musstafa: Optimizers in Deep Learning, 2021.03.27. URL https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0. [Online; accessed 7-December-2022].

[33] Chris Nicholson: A Beginner's Guide to Neural Networks and Deep Learning. URL https://wiki.pathmind.com/neural-network. [Online; accessed 7-December-2022].

[34] Nikon: Nikon Electronic Format (NEF). URL https://www.nikonusa.com/en/learn-and-explore/a/products-and-innovation/nikon-electronic-format-nef.html. [Online; accessed 7-December-2022].

[35] Swarnima Pandey: How to choose the size of the convolution filter or Kernel size for CNN?, 2020.07.23. URL https://medium.com/analytics-vidhya/how-to-choose-the-size-of-the-convolution-filter-or-kernel-size-for-cnn-86a55a1e2c
[Online; accessed 7-December-2022].

[36] PyExifTool — pypi.org, 2022.08.28.
URL https://pypi.org/project/PyExifTool/. [Online; accessed 7-December-2022].

[37] rawpy — API Reference.
URL https://letmaik.github.io/rawpy/api/index.html. [Online; accessed 7-December-2022].

[38] Residual Connection.
URL https://paperswithcode.com/method/residual-connection. [Online; accessed 7-December-2022].

[39] RMSProp — ML Glossary, 2017. URL https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#rmsprop. [Online; accessed 7-December-2022].

[40] Google Scholar: Engineering computer science top publications, 2022.
URL https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng. [Online; accessed 7-December-2022].

[41] Wanshun Wong Towards Data Science.: What is Residual Connection?, 2021.12. URL https://towardsdatascience.com/what-is-residual-connection-efb07cab0d55. [Online; accessed 7-December-2022].

[42] SGD, 2017.
URL .https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#sgd. [Online; accessed 7-December-2022].

[43] Sagar Sharma: Activation Functions in Neural Networks, 2017.09.06. URL https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Online; accessed 7-December-2022].

[44] Sigmoidfunktion — Wikipedia, The Free Encyclopedia, 2022.05.17.
URL https://de.wikipedia.org/w/index.php?title=Sigmoidfunktion&oldid=222949860. [Online; accessed 7-December-2022].

[45] TensorFlow: Module: tf.keras.layers. — TensorFlow API Documentation.
URL https://www.TensorFlow.org/api_docs/python/tf/keras/layers/. [Online; accessed 7-December-2022].

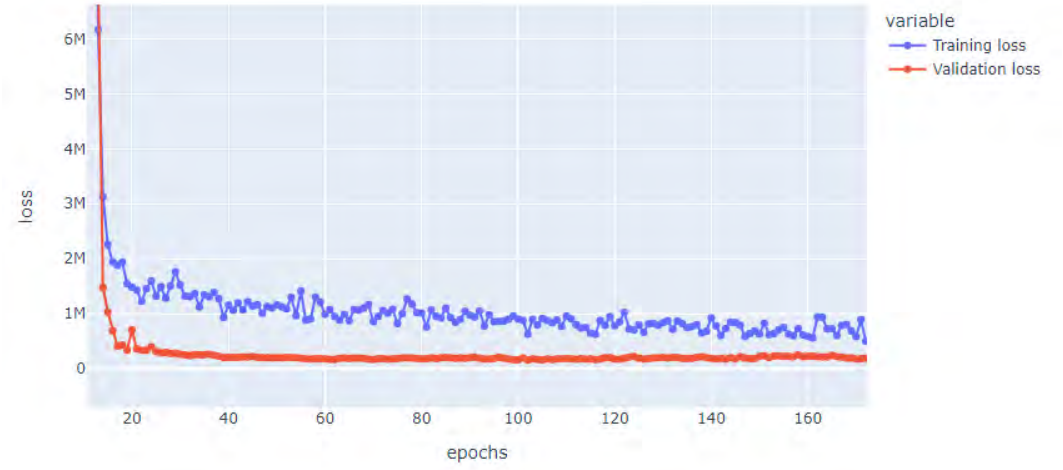
[46] TensorFlow: Save and load Keras models.
URL https://www.TensorFlow.org/guide/keras/save_and_serialize. [Online; accessed 7-December-2022].

[47] TensorFlow: API Documentation, 2021.04. URL https://www.tensorflow.org/api_docs. [Online; accessed 7-December-2022].

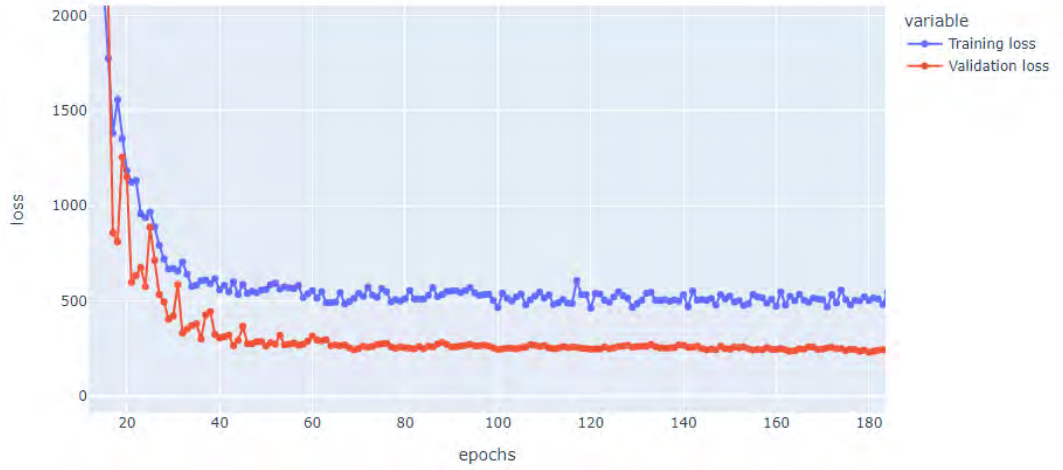
[48] The Model Class — Keras API reference, Models API.
URL https://keras.io/api/models/. [Online; accessed 7-December-2022].

[49] Tutorials: White Balance — Cambridge in Colour. URL https://www.cambridgeincolour.com/tutorials/white-balance.htm. [Online; accessed 7-December-2022].

Függelék



F.0.1. ábra. A *modelNN_init* háló Adadelta optimalizálóval, nagyított hibagörbe



F.0.2. ábra. A *modelNN_irv2* háló Adadelta optimalizálóval, nagyított hibagörbe