FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és

Pongrácz Vince Balázs

Gépi tanulással

Konzulens

BUDAPEST, 2022

Tartalomjegyzék

[Összefoglaló 5](#_Toc120008823)

[Abstract 6](#_Toc120008824)

[1 Bevezetés 7](#_Toc120008825)

[1. Témaválasztás 7](#_Toc120008826)

[2. Témaválasztási szempontok, személyes motiváció 7](#_Toc120008827)

[3. Szakdolgozat felépítése 8](#_Toc120008828)

[2 Mesterséges intelligencia, gépi tanulás 9](#_Toc120008829)

[4. Neurális hálók és mély tanulás 10](#_Toc120008830)

[Konvolúciós neurális hálók (convolutional neural networks, CNN) 12](#_Toc120008831)

[3 Követelmények 16](#_Toc120008832)

[4 Felhasznált technológiák, eszközök 18](#_Toc120008833)

[5 Alkalmazás fejlesztése 20](#_Toc120008834)

[Adatok előfeldolgozása 20](#_Toc120008835)

[A nyers fájlformátumok (.NEF, .CR2, .ARW) 20](#_Toc120008836)

[Az XMP fájlformátum 21](#_Toc120008837)

[Feldolgozás 21](#_Toc120008838)

[5. neurális hálók felépítése, modellek létrehozása 23](#_Toc120008839)

[6. Modellek 25](#_Toc120008840)

[7. Optimalizálók 25](#_Toc120008841)

[8. Hibafüggvények 26](#_Toc120008842)

[6 Modellek leírása és eredményeik elemzése 27](#_Toc120008843)

[7 Munka értékelése, továbbfejlesztési lehetőségek 29](#_Toc120008844)

[Irodalomjegyzék 30](#_Toc120008845)

[Függelék 31](#_Toc120008846)

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 hitelesített 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.

Kelt: Budapest, 2022. 11. 21.

...…………………………………………….

Pongácz Vince

Összefoglaló

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.

# Bevezetés

A következő fejezetben mutatom be röviden a választott témát, a választás szempontjait, személyes motivációm a téma iránt, illetve a szakdolgozat felépítését.

## Témaválasztás

Az utóbbi évtizedben óriási fejlődés figyelhető meg a mesterséges intelligencia és az adattudomány (data science) területén, lényegileg évről évre jönnek ki a jobbnál jobb keretrendszerek és architektúrák, 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 térhódítása, illetve támogató jelenléte. Gondolhatunk itt az autókban az olyan vezetést segítő rendszerekre, mint a sávtartás, táblafelismerés, vagy a nagy közösségi oldalak, videómegosztók ajánlórendszerei, amik eszméletlen pontossággal tudják megtanulni a felhasználók szokásait és pontosan azt a tartalmat a felhasználó elé tenni, aminek következtében a legtöbb időt tölti az adott oldalon, akár függőségbe hajtva és elpazarolva idejét.



<https://trends.google.com/trends/explore?date=all&q=Machine%20Learning,Deep%20Learning>

TODO: Esetleg más statisztika?

Lenyűgöztek korábbi tanulmányaim alatt a mesterséges intelligencia képességei, hogy mennyire pontosak tudnak lenni, mennyire valósan tudnak prediktálni, ezért is kezdtem el mélyebben ezzel a területtel, azon belül is a mély tanulással foglalkozni.

## Témaválasztási szempontok, személyes motiváció

A képfeldolgozás területéről választottam a témám, egy retusálást segítő alkalmazást tűztem ki magamnak célul. Adta magát az ötlet, mivel nagy szenvedélyem a fotózás, évek óta űzöm ezt a hobbit, a SPOT fotókörnek is tagja vagyok. Aki kevésbé jártas a fotózásban azt hiheti, hogy amint lenyomták az exponálógombot és kattant a gép, máris kész a kép, a fotós munkája csak addig a pár ezredmásodpercig tartott. Ez a valóságban sokszor nem így van, ezután jön a fotózás hosszabb része, a képek retusálása.

Magamon is észrevettem, hogy sokáig tart ez a folyamat, néha igen pepecselős munka, hiába vannak már jól bevált presetek, amik nagyjából jóra be tudnak állítani egy képeket, ez a megoldás az összes képre ugyan azokat a beállításokat alkalmazza, pedig egy sorozatban nagyon különböző képek, nagyon különböző beállításokkal is előfordulhatnak, így ez nem egy teljesen járható út.

Ezért jött az ötlet, hogy a gyakrabban változó képparamétereket jó lenne minden képre egyénileg prediktálni, az utómunkázás emberi részét valahogy automatizálni. Ilyen, gyakran képenként változó paraméterek például a fehéregyensúly (white balance, WB), a lila-zöld egyensúly (tint), a világosság (exposure), a kontraszt (contrast), illetve a vibrálás (vibrance). Mivel itt pont a retusálás emberi részének gyorsítása és automatizálása a cél, kézenfekvő választás volt gépi tanulás alkalmazása. Alkalmazásom a fentebb említett paramétereket, azaz a képtulajdonságokat prediktálja a nyersképekből kiindulva, majd ezeket az értékeket képenként korrekciós fájlba írja. Az Adobe Lightroom ezeket a korrekciós fájlokat az importálásánál a képekkel együtt feldolgozza és minden képre a saját korrekcióját alkalmazza, mintha egy előzetes beállítás lenne. A hatás pont olyan, mintha presetet alkalmaztunk volna, csak ez minden képre teljesen egyéni. Az eredmény: automatizáltuk a retusálás egy részét. Apró megjegyzés a nyersképekhez: Ez egy gyártóspecifikus képformátum, ami a kamera nyers szenzoradatait tartalmazza nagyjából teljesen feldolgozatlanul, ezáltal nagy szerkesztési szabadságot hagyva a fotósnak.

## A szakdolgozat felépítése

Szakdolgozatomat az alábbi fejezetekre tagoltam:

1. Bevezetés
2. Mesterséges intelligencia, gépi tanulás, neurális hálók és mély tanulás
3. Követelmények
4. Felhasznált technológiák
5. Az alkalmazás fejlesztése
6. Modellek leírása és eredmények értékelése
7. Munka értékelése, továbbfejlesztési lehetőségek

# Mesterséges intelligencia, gépi tanulás

Az embereket mindig is foglalkoztatta, hogy hogyan lehet a gépeket emberhez hasonló intelligenciával ellátni.

A gépi tanulás ötlete nem ma fogalmazódott meg először, de napjainkban óriási lendületet kapott a temérdek adatnak és a megnövekedett számítási kapacitásnak köszönhetően, ám a matematikai, elméleti alapokat már az 1950-es évektől kezdve elkezdték lerakni.

1956 nyarán Dartmouthban volt egy konferencia (Dartmouth Summer Research Project on Artificial Intelligence), amit a mesterséges intelligencia és a gépi tanulás akadémiai kezdetének tekinthetünk. Ezen a 8 hetes konferencián ültek le és gondolkodtak többek között olyan tudósok, mint Marvin Minsky (), Nathaniel Rochester (), vagy Claude Shannon (), hogy miként kell kinéznie a mesterséges intelligenciának.

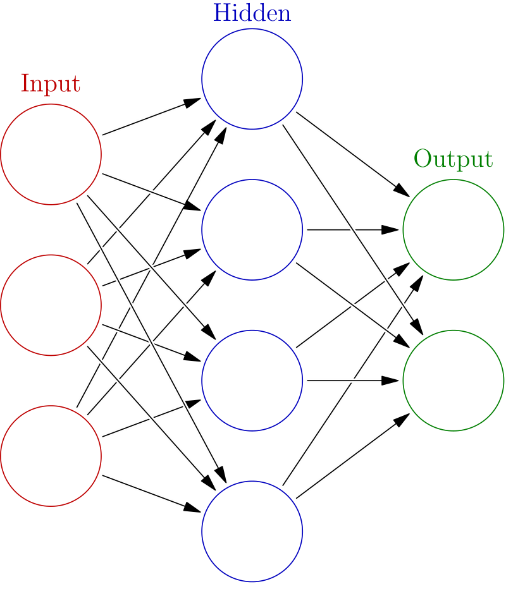
TODO: Ide még valami bevezető szöveg

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.

TODO: ide is még kellene valami

## Neurális hálók és mély tanulás

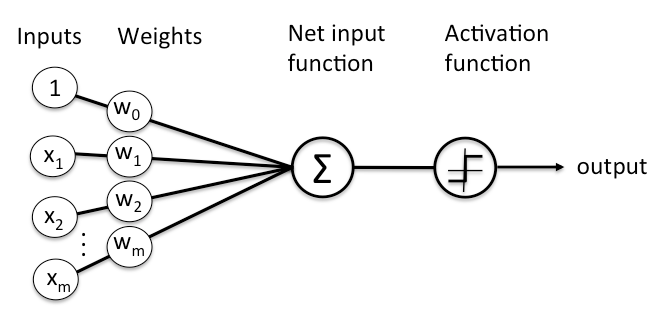
A neurális háló napjainkban az egyik legelterjedtebb gépi tanulási modell. Ez a modell sok hasonlóságot mutat az emberi agyhoz, abban is sok kisebb egység, azaz neuron kapcsolódik egymáshoz, ezáltal egy bonyolult hálót alkotva.



. ábra: Egy egyszerű neurális háló sematikus rajza

Forrás:<https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg>

A neurális hálónak van egy bemeneti rétege (input layer), egy rejtett rétegei (hidden layers) és egy kimeneti rétege (output layer). Ezek a rétegek egyszerű feldolgozóegységekból, 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.



. ábra: Egy neuron felépítése

Forrás: <https://wiki.pathmind.com/neural-network>

A háló kimenete alapján két fajta neurális hálót különböztethetünk meg. Ha a kimenet diszkrét, akkor a neurális háló osztályozási feladatra alkalmas, klasszifikációról (classification) beszélünk. Erre jó példa a feladat, amiben el kell dönteni, hogy kutya vagy macska van egy képen. Amennyiben a kimenet folytonos, azaz végtelen sok értéket vehet fel, akkor regresszióról (regression) beszélünk. Szakdolgozatomban regresszióra 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. Megjegyezném, hogy az aktivációs függvények tanítása (frissítése) is egy járható út lenne, folynak is kutatások erre vonatkozóan. Jelenleg ez a gyakorlatban nem elterjedt, de az 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 optimalizálónak (optimizer), ami a súlyfrissítés ütemét (learning rate) befolyásolja. (<https://arxiv.org/abs/2005.00817>, <https://www.sciencedirect.com/science/article/pii/S0893608021000344> )

Látható, hogy az előre irányát használva a hálónak a kimeneteket kapjuk meg. Amikor a hálót alkalmazni szeretnénk, mert már kielégítő pontossággal adja meg azokat az értékeket, amiket prediktálni szeretnénk, akkor csak ezt az előre irányt használjuk. Amikor a tanítás fázisában vagyunk, akkor fontos a visszajelzés, ezért a kimenetről információt kell visszajuttatni a neuronoknak, ami alapján egy következő minta alkalmával már jobb értékeket fognak prediktálni.

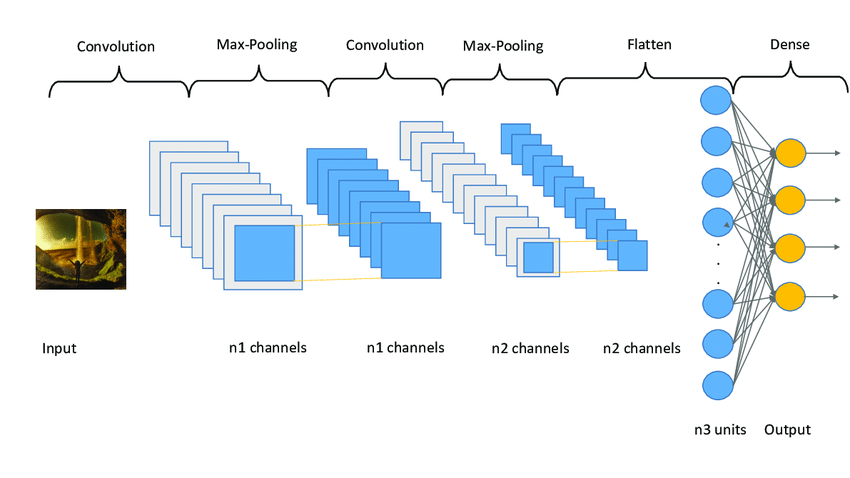
A mély tanulás (deep learning) szintén a neurális hálókra épít, itt a hangsúly a nagyszámú rejtett rétegen van, azaz a hálózat mélysége miatt hívják mély tanulásnak. A több rétegnek előnye a nem neurális háló alapú modellekkel szemben, hogy a képek jellemzői automatikusan kerülnek felismerésre, nem kell külön kézzel az adatban kutatni, hogy mely jellemző befolyásolja azt, amit prediktálni szeretnénk. A megtanult tudás is rétegesen alakul ki, például a képfeldolgozási feladatoknál az alsóbb rétegekben az élek, a sötétebb/világosabb részei a képnek kerülnek felismerésre, míg a későbbi rétegekben konkrétabb alakzatokat ismer fel a háló.

### Konvolúciós neurális hálók (convolutional neural networks, CNN)

A konvolúciós neurális hálókat főleg képfeldolgozásban használják. Működési elvük ugyan az, mint amit fentebb írtam, a különlegesség a többdimenziós adatok kezelésében van.

Képek feldolgozása során egy [x \* y \* z] mátrixot kapunk 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.

A konvolúciós neurális hálók 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 igazából a már korábban tárgyalt, mondhatni normál rétegeket jelenti a neurális hálóban.

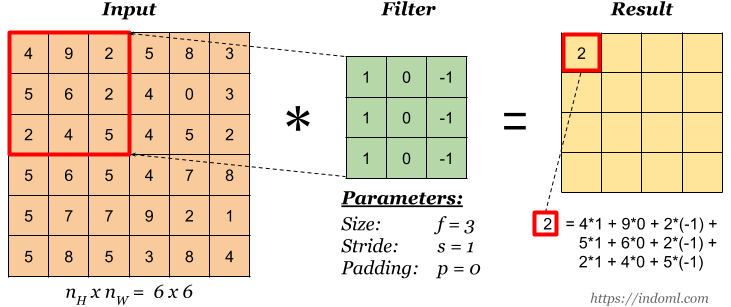


. ábra: Egyszerű konvolúciós neurális háló

<https://www.researchgate.net/publication/339447623_Detecting_Respiratory_Pathologies_Using_Convolutional_Neural_Networks_and_Variational_Autoencoders_for_Unbalancing_Data>

#### 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 összeszorozzuk és összeadjuk. Ez adja ki a következő képréteget, ezt a szakirodalom aktivációs térképnek (activation map) nevezi, amelyre utána további rétegek kerülhetnek, újabb konvolúciós réteg, összevonó, vagy sorosító réteg (flatten layer).



<https://www.projectpro.io/article/introduction-to-convolutional-neural-networks-algorithm-architecture/560>

#### Összevonó réteg (pooling layer)

Ennek a rétegnek a célja az aktivációs térképek dimenziójának csökkentése. 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.



. ábra: Average és Max pooling rétegek példája

A fentebbi összevonásoknak létezik globális változója (global average pooling, global max pooling), ezt a sorosító (flatten) réteg helyett szokták alkalmazni. Ekkor az egész bemenet aktivációs térképenként egy értékre lesz leképezve, azaz az egész bemenet lesz átlagolva, vagy az egész bemenet maximuma, illetve minimuma lesz kiválasztva. Több aktivációs térkép esetén ez egy vektorrá áll össze.

(<http://home.mit.bme.hu/~hadhazi/Oktatas/NN19/Mely_CNN.pdf> )

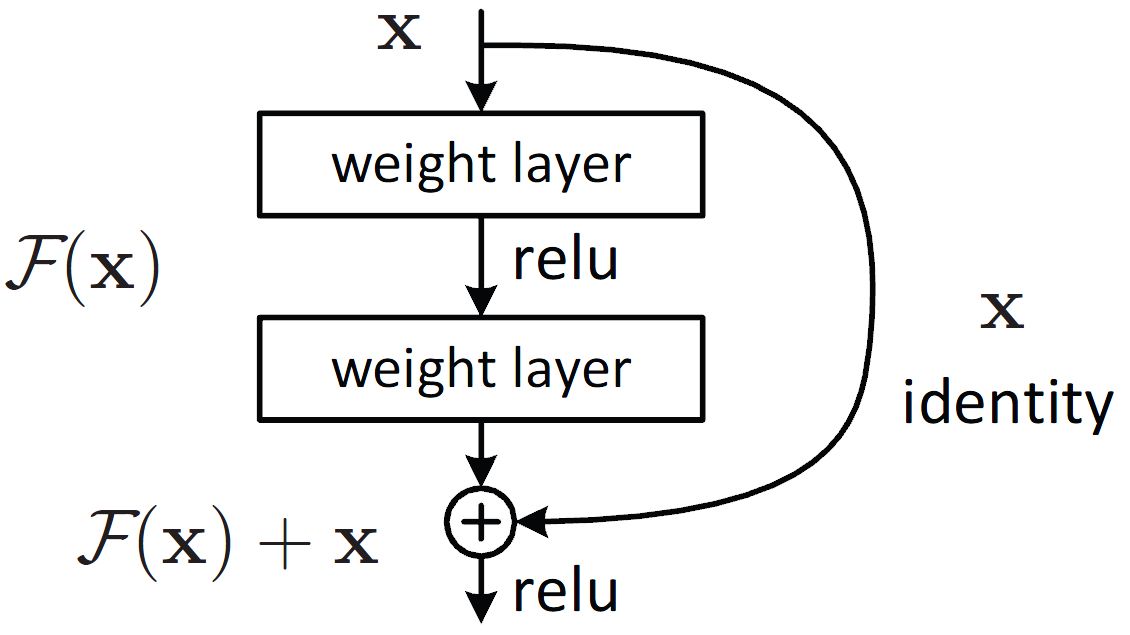
<https://paperswithcode.com/method/average-pooling>

<https://paperswithcode.com/method/global-average-pooling>

#### Residual connections

A residual connection-ök a mély hálózatokban előjövő vanishing/exploding gradients problematikáját próbálják megoldani meg. A probléma gyökere, hogy a tanítási információ vagy elveszik a mélyebb rétegekbe érve, vagy túlságosan nagy értékeket vesz fel, felrobban. 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 residual connection-ön) 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 residual connection-ök segítségével a rétegek kimenete az eredeti bemenet (vagy előző réteg által előállított activation map) és az átugrott rétegekben megtanult kiegészítés összegeként áll elő. Összegzés helyett konkatenálni is szokták a két útvonal értékeit.



. ábra: Residual connection

<https://paperswithcode.com/method/residual-connection>

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, frissítési információ egésze eljuthat az összes réteghez, nem veszik el tanulási információ. A sima előrecsatolt (feedforward) neurális hálóknál a bemenet az összes rétegen keresztülmegy, egy darab útvonal van. A residual connectionnel ellátott hálók esetében számos útvonal van, amin a bemenet a háló végére érhet, több útvonal, azaz több háló együttes eredménye van a végén összegezve, így érthető módon nagyobb a modell komplexitása. Tehát 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.

<https://www.kaggle.com/code/residentmario/notes-on-residual-connections>

<https://towardsdatascience.com/what-is-residual-connection-efb07cab0d55>

# Követelmények

Az alkalmazás célja már előre betanított neurális hálók segítségével még ismeretlen, de a tanítási adathalmazhoz hasonló nyersképek bizonyos tulajdonságainak helyes prediktálása, ezen tulajdonságok .xmp fájlba írása, majd a prediktálás után az Adobe Lightroom program megnyitása, ahol még utólagos korrekciókra van lehetőség, ha a neurális hálók nem találták el jól a képtulajdonságokat.

Az alkalmazás sikerességéhez a következő követelmények teljesülése szükséges:

* Megfelelő minőségű és mennyiségű adat: A mély tanulásban nagyon fontos, hogy nagy számosságú, jó minőségű adathalmaz álljon rendelkezésünkre. Itt akár több milliárd kép is kellene a ténylegesen hatékony és pontos végeredményhez. Sajnos ez a követelmény nem tud teljesülni, mivel nincs a publikus adathalmazok között olyan, ahol ilyen mennyiségben tartanának nyersképeket korrekciós fájlokkal együtt, mivel ez óriási adatmennyiség. Ennek ellenére adataugmentálással (data augmentation) megpróbálok segíteni ezen a problémán.

A nyilvánosan elérhető projektek nagy része kisméretű képek sokaságán hajt végre osztályozási feladatot. A probléma, amivel foglalkozom, 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, rétegek típusán és számán, neuronok számán, tanulási rátán (learning rate), bemeneti adatokon, hibafüggvényeken (loss functions), optimalizálókon (optimizers), tanulási ciklusok (epochs) számán.
* Prediktált tulajdonságok képszerkesztő által feldolgozhatók: A prediktált értékekből .xmp fájlt kell generálni azonos névvel, mint a nyerskép. A Lightroom így amikor importálja a nyersképeket, rögtön alkalmazza a mellette lévő korrekciót is.
* Utólagos finomhangolás a tulajdonságokon képszerkesztőben: Miután minden korrekciós fájl ki lett írva, érdemes a képeket importálni Lightroomba, hogy lássuk a módosítások eredményét, ne csak korrekciós értékeket. Ilyenkor a Lightroom belül lehet még alakítani a képeken, ha ez szükséges.

# Felhasznált technológiák, eszközök

Az alkalmazást python nyelven készítettem el, mert a legtöbb gépi tanulással, adatfeldolgozással és mély tanulással foglalkozó programkönyvtár erre a nyelvre létezik. Kvázi a machine learning magas szintű programozási nyelve mára a python lett, ezért célszerű volt ezt megtanulni. A pythonon kívül még az R-t szokták használni, de oka lehet annak, hogy az eddigi machine learning és data science kurzusok, amiken voltam, ott mindenhol pythonban kellett feladatokat megoldani. Ezen felül a keretrendszerek, könyvtárak is amiket használok pythonban a legteljesebbek.

Fejlesztőkörnyezetnek lokálisan a Visual Studio Code-ot választottam. Kiróbáltam a pycharmot is, az IntelliJ python fejlesztőknek kínált környezetét, de a Code jobb volt, egyszerűbb volt konfigurálni.

Mivel a neurális hálók tanítása nagyon számításigényes művelet, ezért a tanítást nem lokálisan végzem, hanem Google Colab-ban. Ez a Google felhőalapú környezete, amit korábbi machine learning, deep learning témájú tárgyak alkalmával volt alkalmam megismerni. Python és korlátozott mértékben shell kódot is lehet benne futtatni. A platform CUDA képes Nvidia GPU-t is 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 Tesla T4-en sikerül tanítanim, ez bőven elégnek tűnik egyelőre. Minden eszköz megvan benne, ami modellek tanításához szükséges.

A már előre elkészített tanítási adathalmazt Google Drive-ban tárolom, a Colab innen olvassa be az adatokat. A tanítás közben generált statisztikákat és diagrammokat, majd a betanult modelleket szintén Drive-ra mentem ki. A Colabnek ez is óriási előnye, hogy könnyen lehetett a Drive-val összekapcsolni.

A tanításhoz tensorflow-t és keras-t használok. Ezek az könyvtárak kiegészítik egymást, magas absztrakciós szintet biztosítanak adatfeldolgozásra és mélytanulási modellek létrehozására, így teljesen az alapjaitól kell megvalósítanom a különböző neuronokat, rétegeket, hibafüggvényeket és még sok más, alacsony szintű kódot. A tensorflow kezeli a neurális háló GPU-ra ültetést is, az azon való futtatást.

A generált tanulási és jósággörbéket a plotly csomag segítségével rajzoltam ki.

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 parseolására hasznos.

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

Az adatok további feldolgozásában és a tensorflow számára is értelmezhető formára hozásában a pandas és a datasets csomag volt nagy segítségemre.

<https://www.tensorflow.org/api_docs>

# Alkalmazás fejlesztése

Az alkalmazás fejlesztését az alábbi lépésekre bontottam le, ezeket a következő bekezdésekben fejtem ki részletesebben:

* 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 nyersek mellé
* opcionálisan Lightroom meghívása, eredmények megtekintése

### Adatok előfeldolgozása

Az adathalmaz, amin tanítani akarok .NEF (Nikon), .CR2 (Canon), vagy .ARW (Sony) fájlformátumú nyersekből áll. Mindegyikhez tartozik egy korrekciós .xmp fájl ugyan azzal a fájlnévvel a nyerskép mellett.

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

Közös jellemző ezekre a formátumokra, hogy feldolgozás nélküliek, a nyers szenzoradat van beleírva. Általában 12 vagy 14 biten tárolnak el egy színt, így nagyobb a képek dinamikatartománya. A JPG és PNG formátum csak 8 biten tárol egy színt.

Nincs előfeldolgozás, így például a fehéregyensúly, vagy a színárnyalatok teljesen szabadon állítható a retusálás során.

* .NEF (Nikon Electronic Format): A Nikon kamerák nyersformátuma, 12 vagy 14 biten tárol el egy színt. <https://www.nikonusa.com/en/learn-and-explore/a/products-and-innovation/nikon-electronic-format-nef.html>
* .CR2 (Canon Raw 2): A Canon kamerák nyersformátuma, 14 biten tárol el legjobb minőségnél egy színt.

<https://fileinfo.com/extension/cr2>

* ARW (Sony Alpha Raw Digital Camera Image): A Sony kamerák nyersformátuma, 14 biten tárol el egy színt.

### 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 és még sok más adat.

<https://fileinfo.com/extension/xmp>

<https://www.adobe.com/devnet/xmp.html>

### Feldolgozás

Az adatok feldolgozásával a process\_data.py szkript foglalkozik.

Először megtisztítottam az adatokat. Ha egy fájl nem a fentebbi formátumok valamelyikébe tartozott, azt töröltem a könyvtárból. Még ebben a lépésben egy listába tettem a fájlokat, majd levágtam a kiterjesztést. Ilyenkor minden fájlnévből kettőnek kell lennie a listában, hiszen egy nyershez kell egy hozzátartozó .xmp. Ha ez valahol nem teljesül, azok a fájlok szintén törlésre kerültek.

A következő lépésben végigiterálva a könyvtáron beolvastam a nyerseket és a hozzá tartozó .xmp-ket.

A nyersek beolvasásához a rawpy csomagot használtam, ez beolvassa a nyersképet egy rawpy.Rawpy osztályba, amiből utána a postprocess() hívással lehet [h\*w\*c] dimenziós numpy tömbként (de bevezetném itt a mátrix megnevezést erre a struktúrára) lehet letárolni a képet, ahol h a magasság, w a szélesség és c a csatornák száma. Ez a 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 értékeit leskálázom a [0, 1] tartományba, majd átméretezem [133, 200, 3]-as méretre, ahol 133 pixel lesz a képek magassága, 200 pixel pedig a szélessége. Ez a méret még kényelmesen kezelhetőnek tűnik a neurális háló számára. Például az InceptionResNetV2 neurális háló, ami egy fejlett, képosztályozásra használt konvolúciós háló (residual blokkokat és inception architektúrával) is [299, 299, 3] bemeneti dimenziókkal dolgozik. Opcionálisan elmentem a nyersek jpg-be konvertált verzióját, majd az újraméretezett képet hozzáfűzöm a képek listájához.

<https://keras.io/api/applications/inceptionresnetv2/>

<https://letmaik.github.io/rawpy/api/index.html>

Az .xmp fájlok feldolgozásához az exiftool csomagot használom, amivel beolvasom a fájlt, az exiftool ezt parseolja, majd a kiolvasott tageket kulcs-érték párok formájában visszaadja. Korábban kicsit elemezve ezeket az adatokat, megtalálhatjuk benne az 5 tanulni kívánt érték tag-jét, ezeket egy listába tárolom el. Ezt a listát pedig hozzáfűzom a céladatok tömbjéhez.

<https://pypi.org/project/PyExifTool/>

A fentieket követően van két listák listája, ami azonos hosszúságú és párhuzamosan tartalmazza a bemeneti adatokat, illetve az elvárt kimeneteket. Ezt a két listát először egy dictonarybe mentem. Erre a két lista felcímkézése miatt van szükség, a későbbi feldolgozás során így egyszerűbb lesz szétválasztani ezt a két listát. Ez után egy Dataset osztályba rakom el, ez a HuggingFace adathalmazok tárolására alkalmas formátuma, munkám folyamán ez bizonyult a legkönnyebben kezelhetőnek adathalmazok mentésére és betöltésére, a későbbiek folyamán ez volt az az adatformátum, amit a tensorflow is fel tudott később dolgozni. Ez után kimentem az adathalmazt, opcionálisan az aktuális dátummal ellátva.

<https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset>

Ezzel az adatok előfeldolgozása meg is van. Ez lokális művelet volt, természetesen ez is nagyon számításigényes tud lenni, ugyanakkor megéri az erőfeszítés, mert a feldolgozott és kimentett adathalmaz nagyjából 70 nyerskép (darabja 24-30 MB, kameraszenzortól függően) és ugyan ennyi .xmp esetén 28 MB körül van.

## Neurális hálók felépítése, modellek létrehozása

Ezt a részét az alkalmazásnak már a Colab-ben csináltam.

Az első blokkban importáltam a szükséges csomagokat, ezek többnyire a tensorflow és a keras különféle rétegei, modelljei és függvényei, amik az adatok betöltéséhez és a neurális hálók építéséhez kellenek. Inicializáltam a tensorboard-ot, 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 ezt szétválasztom 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 képmátrixhoz 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, ezáltal robosztusabb és stabilabb lesz. Az augmentáció előfeldolgozó (preprocessing) rétegekkel valósul meg. Ezek közül a RandomFlip-et és a RandomRotation-t használtam fel. Ezek véletlenszerűen horizontálisan tükrözik és forgatják bizonyos szögtartományok között a bemeneti képet.

Mivel a jellemzők közül néhány egészen eltérő értéktartományban van, 5 külön neurális hálót tanítok be, tehát az elvárt kimeneteket szét kell választani az 5 hálónak megfelelően.

Felhasznált keras rétegek:

* Dense: teljesen normális fully connected réteg a neurális hálóban
* 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.
* Activation: Réteg az aktivációs függvénynek.
* Flatten: Sorosító réteg, ami kilapítja a bemenetet és egy vektort hoz létre belőle. A bemenet itt célszerűen mátrix formában van, például egy konvolúciós réteg kimenete. Nem tartalmaz tanítható paramétereket, hiszen csak egy egyszerű átdimenzionálás.
* 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 képméretet 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, hosszú ideig is tartott a tanítás. Akkor még nem volt meg feltétlenül a mély hálózatok tanításához szükséges számítási kapacitás, amivel ki lehet váltani a nagyobb rétegeket.

<https://medium.com/analytics-vidhya/how-to-choose-the-size-of-the-convolution-filter-or-kernel-size-for-cnn-86a55a1e2d15>

* MaxPooling2D: Maximum összevonó réteg, itt a kernel méretét kell megadni. Régebben szintén használtak nagyobb filtereket, ma a 2x2-es és a 3x3-as a legelterjedtebb. Ez a réteg az előző réteg méretét csökkenti és nyeri ki belőle a legrelevánsabb információt.
* GlobalAveragePooling2D: Globális összevonó réteg, az egész aktivációs térkép átlagát veszi, ezt az egy értéket adja tovább. Sorosító réteg helyett célszerű használni.
* Add: Ez a réteg több réteg összevonására szolgál, amiknek azonos a kimeneti dimenziója. A residual connection-öknél használom, ahol konvolúciós rétegeket és a bemenetet összegzi.
* 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 vagy vízszintesen.

<https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomFlip>

* RandomRotation: Szintén adataugmentációs réteg, a bemeneti képen végez véletlenszerű forgatást.

<https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomRotation>

A modelleket felépítő függvények deklarációja a könnyebb kezelhetőség érdekében egyformák, eltérés csak a default 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 eső 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 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-et adtam meg az első generációs hálózatoknak, azaz a neuronok közötti kapcsolatok 0.1 része 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 segíti a modell tanulását, stabilabb és gyorsabban tanul a modell batch normalizálással, mint nélküle.

<https://en.wikipedia.org/wiki/Batch_normalization>

<https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338>

A name paraméter értelemszerűen a modell neve lesz.

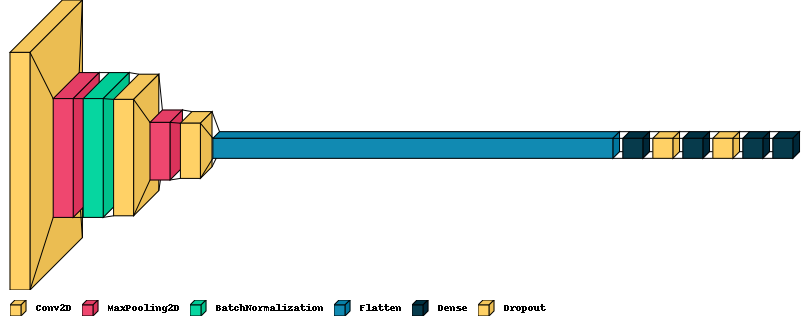
Mindegyik modellépítő függvény egy Sequential példányra kezdi el építeni a rétegeket és egy ilyen példánnyal is tér vissza. Ez az osztály alkalmas a rétegek egymásra építéséhez, egységesíti ezeket a rétegeket egy modellé.

<https://keras.io/api/models/sequential/>

## Modellek létrehozása

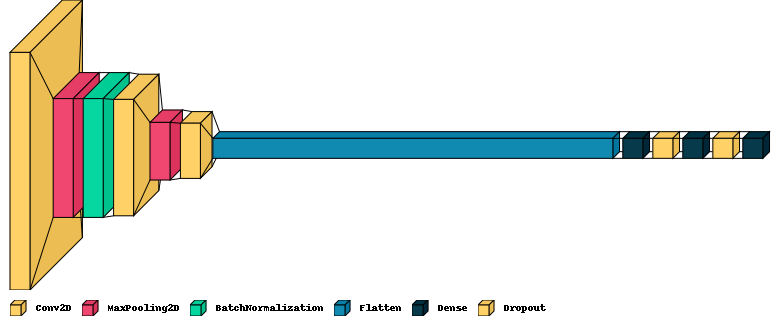
Mindegyik modellnek van egy saját modellépítő függvénye, ami létrehozza az adott hálózatot.

buildNN\_init: egy sima konvolúciós hálózatot hoz létre. Felépítése:



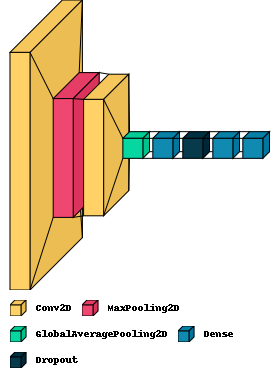
6. ábra: A buildNN\_init architektúra felépítése

buildNN\_s: Az előző hálóhoz hasonló, ugyan annyi rétege van, csak kevesebb neuronból állnak a rétegek



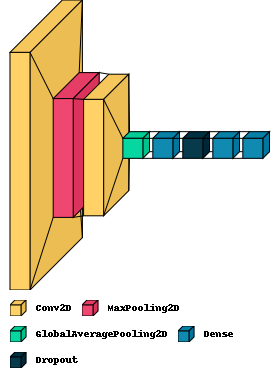
7. ábra: A buildNN\_s architektúra felépítése

buildNN\_xs: A buildNN\_s-hez képest kevesebb rétegből áll. Flatten helyett GlobalAveragePooling2D a sorosító réteg.



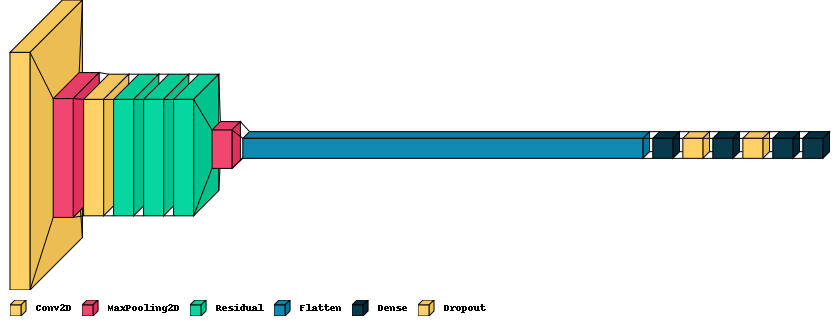
8. ábra: A buildNN\_xs modell felépítése

buildNN\_xxs: A buildNN\_xs-hez képest a változás, hogy kevesebb neuron alkot egy réteget.



9. ábra: A buildNN\_xxs modell felépítése

buildNN\_RCNN: A buildNN\_init-ből indult, a konvolúciós rétegek között residual connection-ök vannak.



10. ábra: A buildNN\_RCNN architektúrája

buildNN\_pretrained: Az InceptionV3 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, az InceptionV3 az a Functional rész.



11. ábra: a buildNN\_pretrained architektúrája

A bemenet természetesen újra lett méretezve, illetve a transfer learning miatt az utolsó rétegek (final part az ábrán) kimaradtak, helyette a fent is látható rétegek jöttek.

<https://paperswithcode.com/method/inception-v3>



12. ábra: Az InceptionV3 architektúrája

### Transfer learning:

Ha kevés adat áll rendelkezésre, de 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 befagyasztjuk, nem taníthatóra állítjuk. Ezután a tetejére a saját rétegeket csatlakoztatunk, ezeket viszont taníthatóra állítjuk. Hagyjuk így tanulni a saját adathalmazunkon. Amikor már elég stabil ez a hálózat, akkor az alaphálón is tanítjuk egy kicsit a modellt, így az alsóbb rétegeket finomhangoljuk.

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

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

Ilyen paraméterek például:

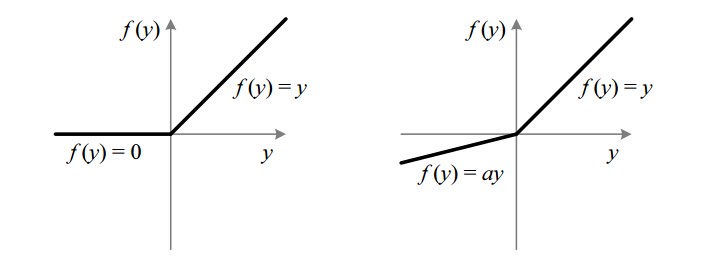
* neuronok száma (hány neuron van egy rétegben, mekkora a konvolúciós, összevonó filter)
* rétegek száma (hány réteg van egymás után, hány konvolúciós filtert teszünk a hálóba)
* dropout alkalmazása (van e a hálóban dropout réteg és ha van akkor a kapcsolatok hányadrésze lesz véletlenszerűen elhagyva)
* batch size (egyszerre hány mintára számolja a háló a hibát)
* tanulási ciklusok (epochs) (hányszor megy át a teljes adathalmaz a hálózaton, vagyis hány cikluson keresztül tanul a hálózat)
* háló architektúrája (van e residual connection a hálóban, inception network; összevonó réteg esetén (pooling layer) átlag vagy max pooling réteg a felhasznált réteg, esetleg egy előre betanított háló tetejére építünk hálót (transfer learning))
* aktivációs függvények (milyen aktivációs függvények vannak a hálóban)
* hibafüggvény (több fajta van, kifejtésre kerülnek)
* optimalizálók (tanulási rátát befolyásol, azaz a tanulás sebességét)

A következő részben felsorolom a kipróbált és felhasznált aktivációs függvényeket (activation functions), hibafüggvényeket (loss functions) és optimalizálókat (optimizers).

### Aktivációs függvények (activation functions)

A neuron kimenetét szabályozzák, a gyakorlatban az alábbi két aktivációs függvény terjedt el egyszerűségük és hatékonyságuk 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 a paraméterrel szorzott bemenet.



. ábra: ReLU és Leaky ReLU

Az első neurális hálók sokat használták aktivációs függvényként a sigmoidot. Ezt lehetne használni, de könnyen szaturálódik, ezért nem célszerű. Szaturálódásnak azt nevezzük, ha nagy x-beli eltérés esetén a függvény értékében kicsi az eltérés, ez az alábbi ábrán jól látható. A gyakorlatban a legtöbb regressziós problémánál a ReLU jön elő.



14. ábra: A sigmoid függvény képe

<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

### Hibafüggvények (loss functions)

* Mean Square Error (MSE): Hiba négyzetének átlaga, ez a legáltalánosabb regressziós problémák esetén.
* Root Mean Square Error (RMSE): Az MSE négyzetgyöke, érzékeny a kiugró hibákra.

<https://en.wikipedia.org/wiki/Root-mean-square_deviation>

* Mean Square Logarithmic Error (MSLE): Az elvárt és prediktált érték hányadosának logaritmusát kell venni, majd az összes mintára átlagolni.

<https://insideaiml.com/blog/MeanSquared-Logarithmic-Error-Loss-1035>

### Optimalizálók (Optimizers)

Az optimalizálók olyan algoritmusok, amik a súlyok frissítésének helyes mértékéért felelnek. A háló tanítása során az a célunk, hogy olyan értékekkel frissítsük a háló súlyait, hogy a hibafüggvény értéke csökkenjen, minimalizáljuk a hibát. A hibafüggvény megadja a háló hibáját, ezt a súlyok szerint deriválva megkapjuk a frissítés mértékét, azaz egy gradienst. A tanulás gyorsítása érdekében ezeket a gradienseket még egy tanulási rátának nevezett paraméterrel szokták megszorozni a gyorsabb konvergencia érdekében. Az optimalizáló algoritmusok ezeket a tanulási rátát, vagy rátákat módosítják, adott esetben adaptívan a gradiens nagyságához, vagy az adott dimenzió frissítéséhez képest.

Az alábbi optimalizálókat használtam fel a neurális hálókhoz, ezek a tf.keras.optimizers. package-ben 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.

<https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#rmsprop>

* 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.

<https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adagrad>

* Adadelta (adaptive delta): Az Adagrad csúszóablakos változata. Jobb, mint az Adagrad, mert akkor is képes még javulni (ezáltal a háló pedig tanulni), amikor az Adagrad már nem. Ennek oka a Adagradnál alkalmazott gradiensakkumulálás.

<https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adadelta>

* Adam (Adaptive Moment Estimation): Az RMSProp és a Momentum ötleteit kombinálja. Minden paraméterhez (jelen esetben súlyhoz) saját adaptív tanulási rátát számol és figyelembe veszi a korábbi gradienseket, azok négyzetét, korrigálja ezeket és ez alapján frissíti a paramétereket.

<https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adam> <https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#momentum>

<https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0>

Az SGD-t (Stochastic Gradient Descent) is használhattam volna, ám még az első hálók tanítása során kiderült, hogy nem elég hatékony, a fentebbi optimalizálók sokkal jobban teljesítettek. Ez az optimalizáló mindig véletlenszerűen választ egy mintát, majd ennek a hibából számolt gradiensét szorozza meg egy előre meghatározott alfa paraméterrel, majd ezt kivonja a súlyokból. Ennek egy továbbfejlesztése, amikor véletlenszerűen választ ki több mintát (batch-et), és a gradiensek átlagával frissíti a súlyokat.

<https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#sgd>

Nincs más hátra, mint hogy kipróbáljuk a modelleket, illetve javítsunk teljesítményükön.

# Modellek leírása és eredményeik elemzése

TODO: Fit models, Plot results, learning curves, predictions

TODO: Ide a modellek tanulási grafikonjai, részletes leírásuk jön (Felépítésük, azonos modell más paraméterezéssel futtatása)

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 dologra kell figyelni és a paramétereket ezek szerint beállítani, ezek a túltanulás (overfitting) és az alultanulás (underfitting).

**Overfitting:**

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

**Underfitting:**

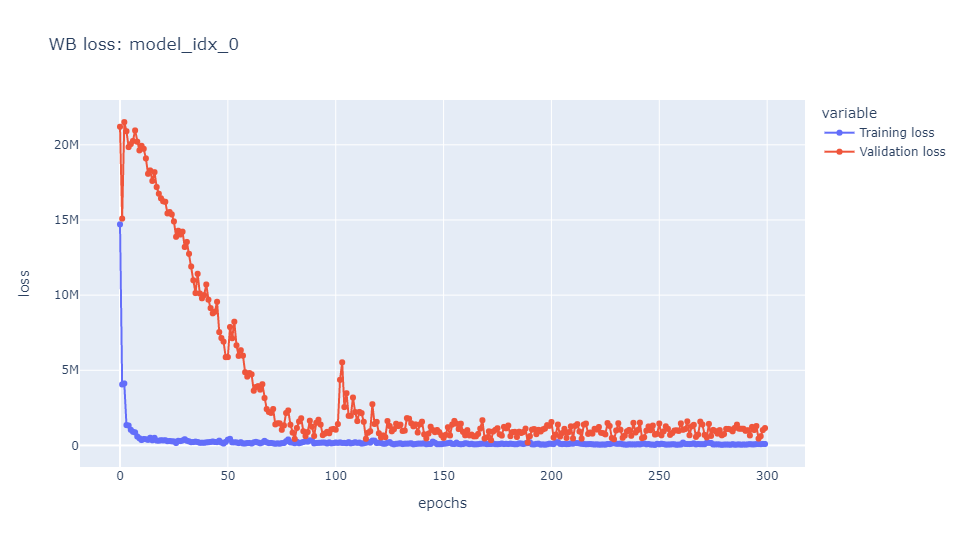
A modell nem elég komplex ahhoz, hogy megtanulja az adathalmazban fellelhető mintázatokat, képtelen elég jó eredményt elérni a validációs halmazon. Ezen kívül a tanulási görbén a tanulási halmaz eredményeinél is az látszik, hogy nem képes a modell tovább javulni és alacsonyabb hibával prediktálni.

A tanuláshoz felhasznált adatok:

* Konstanz adathalmaz: Az adathalmaz egy nyári, konstanzi kirándulás képeiből áll, összesen 56 kép a tanulási halmaz, de ez a duplájára nőtt az augmentáció miatt. A tesztelési halmazban 12 kép található, ugyan ebből az adathalmazból.
* Szakestély adathalmaz: Ez az adathalmaz az egyik Villanykari Szakestély képeiből áll, ahol a SPOT fotókörrel fotóztam. 136 képből áll, augmentálva 272 kép. A tesztelési halmaz x kép.

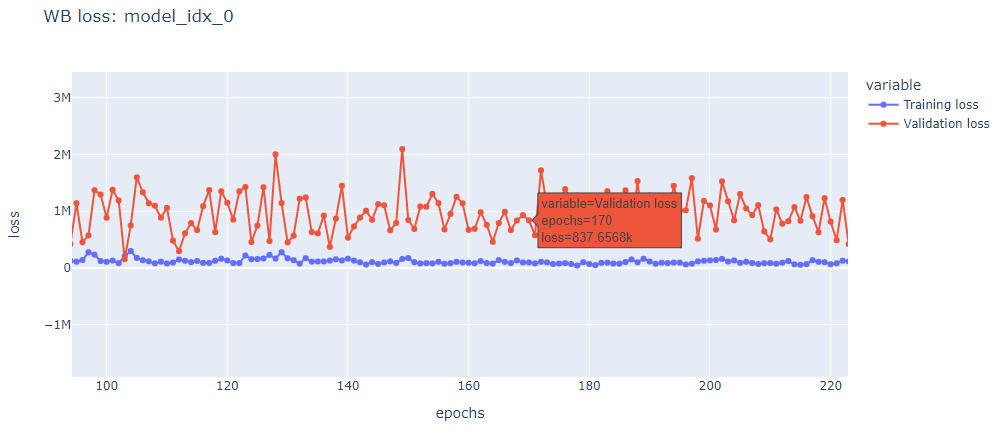
A modellválasztást a fent definiált 5 architektúra vizsgálatával kezdtem egy alap paraméterezéssel a Konstanz adathalmazon. Minden háló 300 tanulási ciklust, 12-es batch méretet, Adam optimizert és MSE loss function-t használ. Első lépésben mind az 5 háló egyetlen kimeneti paraméterre, a fehéregyensúlyra tanul, így segítve a különböző modellek közötti teljesítménybeli különbségek észrevételét.

Az alábbi ábrán az első, buildNN\_init-tel felépített háló tanulási görbéje látható. Ezen, de a következő grafikonokon is az x tengelyen a tanulási ciklusok (epochs) száma, az y tengelyen a batch-enként számolt hiba értéke látszik, mind a tanulási (training set), mint a validációs halmazon (validation set).



Ezen a grafikonon a tanulási hiba folyamatosan csökken, míg a validációs hiba egy darabig csökken, de utána nem mutat elégséges stabilitást, nem tudja elég hatékonyan megközelíteni a tanulási halmazon elért teljesítményt. Látszik, hogy 150 ciklus után nem történik érdemi változás.

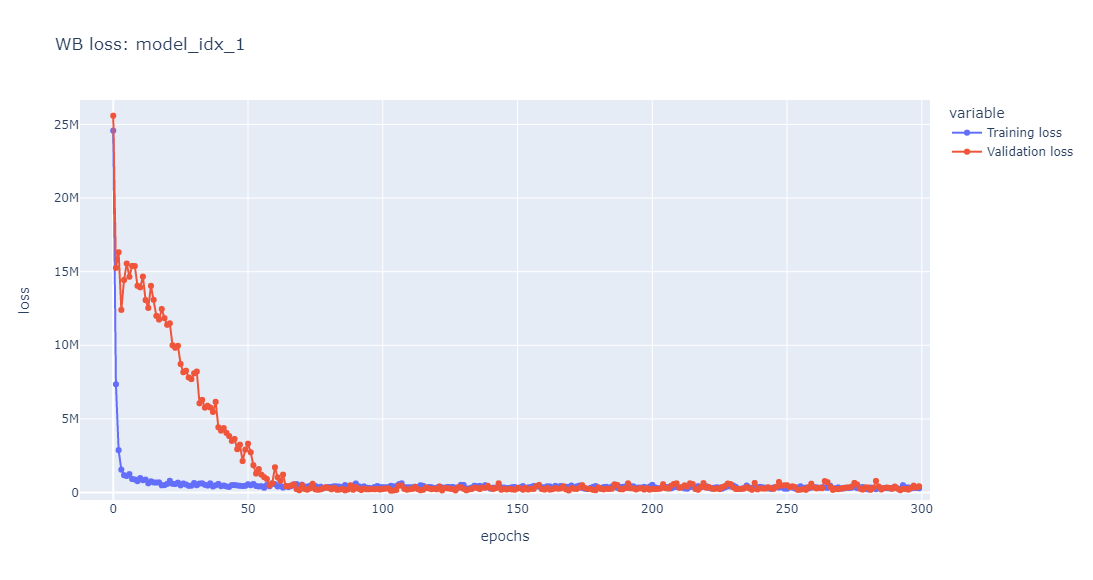
Az következő grafikon az előző tanulási görbe egy kinagyított szelete, nagyjából 100 epoch-os intervallumot ölel fel. A validációs hiba 800.000 körül mozog, a tanulási hiba viszont a 150.000-es tartomány. Egy jó modellnek ennél jobban meg kellene közelíteni a tanulási hibát.



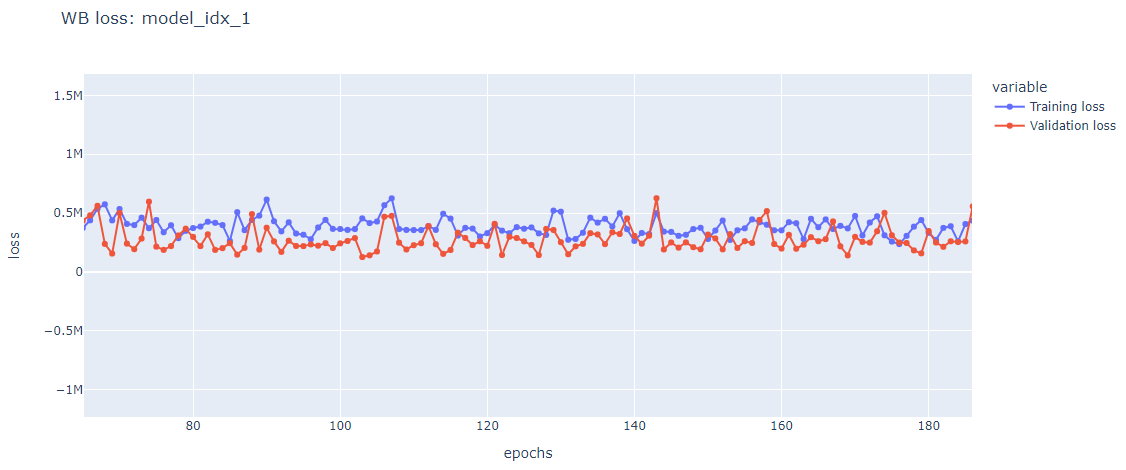
A probléma az optimalizálókra (optimizers) és a tanulási rátára (learning rate) vezethető vissza, mivel az látszik, hogy a háló megrekedt egy olyan állapotban, ahonnan nem képes további érdemi fejlődésre. Ennek megoldása a tanulási ráta másfajta inicializálása, illetve tanulás közbeni másfajta frissítése.

<https://analyticsindiamag.com/what-is-the-plateau-problem-in-neural-networks-and-how-to-fix-it/>

A következő háló a buildNN\_s függvénnyel lett létrehozva, ez kevesebb rétegből áll mint a buildNN\_init háló. Tanulási görbéje következőképp néz ki:



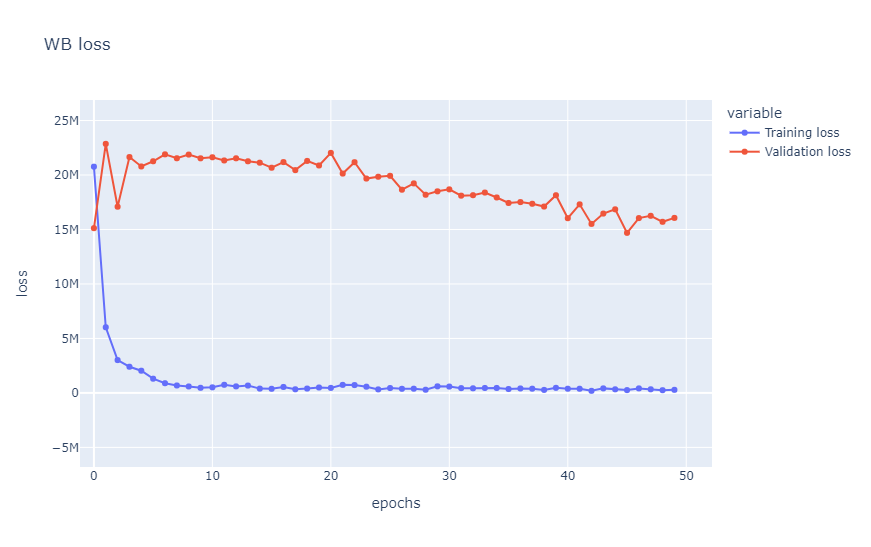
Az érdekesebb részekről egy nagyítás:



A fentebbi grafikonon sokkal bíztatóbb eredményeket lehet megfigyelni. A háló képes volt arra, hogy a validációs halmazon is hasonlóan jó eredményt érjen el, mint a teszthalmazon, számtalanszor van a validációs halmaz hibaértéke a teszthalmaz hibáival egy szinten. Az előbbi plató viszont itt is megfigyelhető, sajnos a tanulási ciklusok végefelé

Ez több okból lehetséges.

Az alábbi képen a fehéregyensúlyt tanulni próbáló buildNN\_init konfiguráció alapján létrehozott háló teljesítményét láthatjuk, nem túl a



…

### Betanult modellek mentése, betöltése

A tanult modelleket HDF5 fájlokként (.h5 kiterjesztés), opcionálisan dátummal ellátva lementem Drive-ra, ezt végzi a save\_models függvény, a tf.keras.Model osztálynak van a mentéshez és a betöltéshez is beépített függvénye. Ezeket a fájlokat utána lokálisan használom majd tovább, ahol a load\_models függvény tölti vissza őket egy listába.

<https://www.tensorflow.org/guide/keras/save_and_serialize>

### Alkalmazás 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. Ezen kívül be kell állítani a Lightroomban az Autoimport funkciót egy tetszőleges, még üres mappára. Ez ahhoz szükséges, hogy a Lightroom automatikusan fel tudja olvasni azokat a képeket, amikhez a korrekciós fájlok készültek. Érdemes az LR környezeti változóban eltárolni a Lightroom.exe-re mutató útvonalat, ez általában így néz ki: `C:\Program Files\Adobe\Adobe Lightroom Classic\Lightroom.exe`. Itt előfordulhat, hogy újra kell indítani a számítógépet.

Le kell tölteni a feltanított modelleket és egy új mappába kell őket másolni, ahol csak az éppen használni kívánt modellek vannak.

### Jellemzők prediktálása

A projekt ezen része az, amit remélhetőleg gyakran fog használni a felhasználó. A predictor.py egy CLI alkalmazás, ami az alábbi paramétereket várja:

**Kötelező paraméter(ek):**

raw\_source\_dir: A prediktálni kívánt nyersképek elérési útja

**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.

**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 aztán minden képhez legyártja a generate\_xmp\_result hívás segítségével a korrekciós fájlokat.

Az XMP fájlformátumhoz sajnos nem találtam kézreeső szerializáló könyvtárat, amit találtam az csak Linux rendszereken fut. A Lightroom viszont csak Windows rendszeren fut. Ezért úgy döntöttem, hogy ezt az .xmp részletet, amely nagyjából 20 sorból áll, stringként írom egy .xmp kiterjesztésű fájlba. A tapasztalat azt mutatja, hogy a Lightroom ezt is szépen, probléma nélkül fel tudja dolgozni és alkalmazni tudja a leírt módosításokat.

Végül, ha 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.

A háló által prediktált eredmények ekkor rögtön láthatók, ha valamit nem sikerült jól eltalálnia a hálónak, akkor a Lightroomban ez kézzel javítható.

<https://stackoverflow.com/questions/60390707/how-to-choose-the-number-of-convolution-layers-and-filters-in-cnn>

<https://machinelearningmastery.com/improve-deep-learning-performance/>

Ha már megvan az adathalmaz, ami megfelelően van felépítve és előkészítve egy neurális háló számára, utána jön a különböző fajta neurális hálók, architektúrák tanítása, kiértékelése.

A jónak ígérkező modelleket ezután

**A GPU computation előnyei:**

A neurális hálózatok tanítása nagyon erőforrásigényes feladat, rengeteg mátrixművelet elvégzésére van szükség. A GPU-k pedig pontosan ebben nagyon jók, gyorsan tudnak mátrixműveleteket végezni.

Míg CPU-n futtatva a tanítást, nagyjából 8-9 másodperc volt egy tanulási ciklus (epoch) tanítási ideje, addig GPU-n futtatva 40-50 milliszekundum volt ugyan ez, így 8 másodperces CPU-s és 40 milliszekundumos GPU-s időt véve is 200-szor gyorsabb volt esetemben a GPU-n tanítás egy tanulási ciklust nézve. Ez persze nagyban függ a Google Colab aktuális leterheltségétől, ugyanakkor jól szemlélteti a GPU-k hatásosságát és megkerülhetetlenségét a mély tanulásban.

---

Notes:

Témaválasztási szempontok, motiváció

Felépítés, struktúra

Feladatértelmezés

Tervezés célja

Feladat indokoltsága

Felépítés összefoglalása

Machine Learning – tensorflow&keras API ismertetése

# Munka értékelése, továbbfejlesztési lehetőségek

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék