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

Képfeldolgozási Algoritmusok Alkalmazása és Vizsgálata

Neurális hálókkal

Konzulens

BUDAPEST, 2022

Tartalomjegyzék

[Összefoglaló 7](#_Toc121176643)

[Abstract 8](#_Toc121176644)

[1 Bevezetés 9](#_Toc121176645)

[1.1 Témaválasztás 9](#_Toc121176646)

[1.2 Témaválasztási szempontok, személyes motiváció 10](#_Toc121176647)

[1.3 A szakdolgozat felépítése 11](#_Toc121176648)

[2 Mesterséges intelligencia, gépi tanulás 12](#_Toc121176649)

[2.1 Gépi tanulás 12](#_Toc121176650)

[2.2 Neurális hálók és mély tanulás 13](#_Toc121176651)

[2.3 Konvolúciós neurális hálók 15](#_Toc121176652)

[2.3.1 Konvolúciós réteg (convolutional layer) 16](#_Toc121176653)

[2.3.2 Összevonó réteg (pooling layer) 17](#_Toc121176654)

[2.3.3 Residual connections 18](#_Toc121176655)

[3 Az alkalmazás és modellek követelményei 20](#_Toc121176656)

[4 Felhasznált technológiák, eszközök 22](#_Toc121176657)

[5 Az alkalmazás fejlesztése 25](#_Toc121176658)

[5.1 Adatok előfeldolgozása 25](#_Toc121176659)

[5.2 A nyers fájlformátumok (.NEF, .CR2, .ARW) 25](#_Toc121176660)

[5.3 Az XMP fájlformátum 26](#_Toc121176661)

[5.4 Feldolgozás 26](#_Toc121176662)

[5.5 Neurális hálók implementálása 28](#_Toc121176663)

[5.6 Modellek létrehozása 31](#_Toc121176664)

[5.7 Transfer learning 34](#_Toc121176665)

[5.8 Modellek specifikálása 35](#_Toc121176666)

[5.8.1 Aktivációs függvények (activation functions) 36](#_Toc121176667)

[5.8.2 Hibafüggvények (loss functions) 37](#_Toc121176668)

[5.8.3 Optimalizálók (Optimizers) 37](#_Toc121176669)

[6 Modellek leírása, eredményeik elemzése, modellválasztás 40](#_Toc121176670)

[6.1 Overfitting, underfitting 40](#_Toc121176671)

[6.2 Adathalmazok 40](#_Toc121176672)

[6.3 Modellválasztás 41](#_Toc121176673)

[6.3.1 Szekvenciális konvolúciós modellek 41](#_Toc121176674)

[6.3.2 Visszacsatolt háló 45](#_Toc121176675)

[6.3.3 Transfer learning 46](#_Toc121176676)

[6.3.3.1 InceptionV3 transfer 46](#_Toc121176677)

[6.3.3.2 MobileNetV2 transfer 48](#_Toc121176678)

[6.3.4 Összefoglaló a modellek hibaértékeiről 49](#_Toc121176679)

[6.3.5 Még egy visszacsatolt konvolúciós háló 50](#_Toc121176680)

[6.3.6 Összehasonlítás különböző adathalmazokon 51](#_Toc121176681)

[6.3.7 Különböző modellparaméterezések 52](#_Toc121176682)

[6.3.7.1 Hibafüggvények: 53](#_Toc121176683)

[6.3.7.2 Optimalizálók 53](#_Toc121176684)

[6.3.7.3 Batch méretek 56](#_Toc121176685)

[6.3.8 Végleges hálók a tulajdonságok prediktálására 56](#_Toc121176686)

[6.4 Betanult modellek mentése, betöltése 58](#_Toc121176687)

[6.5 Alkalmazás előtti konfigurációk 59](#_Toc121176688)

[6.6 Jellemzők prediktálása 59](#_Toc121176689)

[6.6.1 Kötelező paraméter(ek): 59](#_Toc121176690)

[6.6.2 Opcionális paraméterek: 59](#_Toc121176691)

[6.6.3 Az alkalmazás futása 60](#_Toc121176692)

[7 Munka értékelése, továbbfejlesztési lehetőségek 63](#_Toc121176693)

[8 Köszönetnyilvánítás 65](#_Toc121176694)

[9 Irodalomjegyzék 66](#_Toc121176695)

[10 Függelék 67](#_Toc121176696)

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. 30.

...…………………………………………….

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

### Témaválasztás

Az utóbbi évtizedben óriási fejlődés figyelhető meg a mesterséges intelligencia (machine learning) 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, táblák felismerése, vagy akár a nagy közösségi oldalak, videómegosztók ajánlórendszerei, amik eszméletlen pontossággal tudják 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 alábbi táblázat a Google Scholar-on Engineering & Computer Science témában fellelhető legnépszerűbb publikációk listájának első 10 elemét mutatja, ezek közül csak cím alapján legalább 5 olyan, ami a mesterséges intelligencia, gépi tanulás, vagy mély tanulás témájába esik. Ez is azt mutatja, hogy a mesterséges intelligencia jelenleg is nagyon gyorsan és dinamikusan fejlődő területe az informatikának, ezért mindenképp érdemes vele foglalkozni.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

. ábra: A Google Scholar Engineering & Computer Science kategória legnépszerűbb publikációi (Google, 2022)

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

### Témaválasztási szempontok, személyes motiváció

A képfeldolgozás területéről választottam a témát, egy retusálást segítő alkalmazást tűztem ki magamnak célul. 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. 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, pedig egy sorozatban nagyon különböző képek, nagyon különböző beállításokkal is előfordulhatnak, így ez nem egy teljesen járható és kényelmes 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), 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 a cél, ehhez célszerű választás volt gépi tanulás alkalmazása. Alkalmazásom a fentebb említett képtulajdonságokat prediktálja a nyersképekből kiindulva, majd ezek az értékek képenként korrekciós fájlokba kerülnek. Az Adobe Lightroom ezeket a korrekciós fájlokat az importálásánál a képekkel együtt feldolgozza és minden képre a saját korrekcióját alkalmazza, mintha egy előzetes képre szabott beállítás lenne. A hatás olyan, mintha presetet alkalmaztunk volna, csak ez így minden képre teljesen egyéni, végeredményként pedig a retusálás egy része automatizált.

### A szakdolgozat felépítése

A szakdolgozatomat az alábbi fejezetekre tagolódik:

1. Bevezetés
2. Mesterséges intelligencia, gépi tanulás, neurális hálók és mély tanulás
3. Az alkalmazás és modellek követelményei
4. Felhasznált technológiák ismertetése
5. Az alkalmazás fejlesztése
6. Modellek ismertetése 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, így a gépi tanulás ötlete sem 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 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) (Wikipedia - Marvin Minsky, 2022), 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) (Wikipedia - John McCarthy, 2022), vagy Claude Shannon (információelmélet atyja, az AI területén a Shannon’s Mouse-t érdemes említeni: egy mechanikus robotegér, ami megtalálja a kiutat a labirintusból) (Wikipedia - Claude Shannon, 2022) találkoztak és gondolkodtak arról, miként kell kinéznie a mesterséges intelligenciának. Többen közülük később maradandót alkottak a mesterséges intelligencia területén is.

### 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, 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 elvárt kimenetekből áll, a cél pedig a bemenet-kimenet párokból megtanulni egy általánosítást, ami a még látatlan bemenetekre is teljesül.

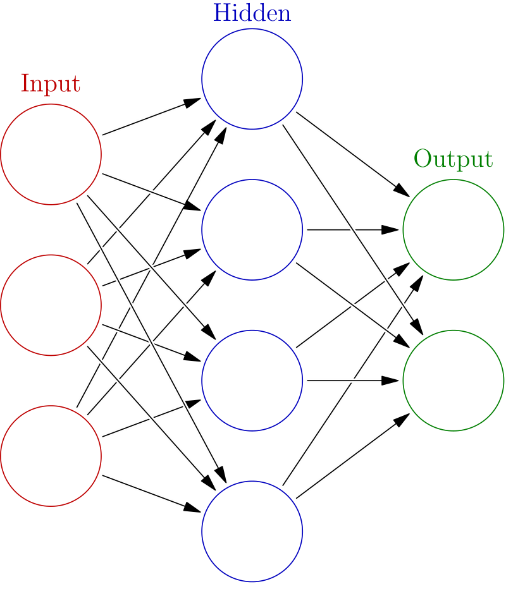
Unsupervised learning esetében a bemeneti adatokhoz nem tartozik elvárt kimenet, itt az adatokban fellelhető struktúrát, mintázatokat kell felismerni, majd ez alapján lehet szabályokat kikövetkeztetni.

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.

### Neurális hálók és mély tanulás

(neural networks and deep learning)

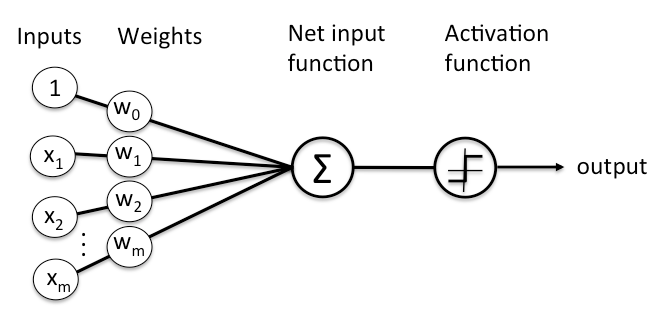
A neurális háló napjainkban az egyik legelterjedtebb gépi tanulási modell. Használattól függően megvalósíthatnak supervised és unsupervised learning-et is, a szakdolgozatban supervised learning-ről lesz szó. A neurális hálók modellje sok hasonlóságot mutat az emberi aggyal, abban is sok kisebb egység,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



. 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), 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 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 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 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 (learning rate) befolyásolja.

<https://arxiv.org/abs/2005.00817>

<https://www.sciencedirect.com/science/article/pii/S0893608021000344>

A hálók előre irányát használva a kimeneteket kapjuk meg. Amikor a hálót már élet 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 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, vagy a sötétebb/világosabb részei a képnek kerülnek felismerésre, míg a későbbi rétegekben konkrétabb alakzatokat ismer fel a háló, ha például alakzatfelismerés a feladat.

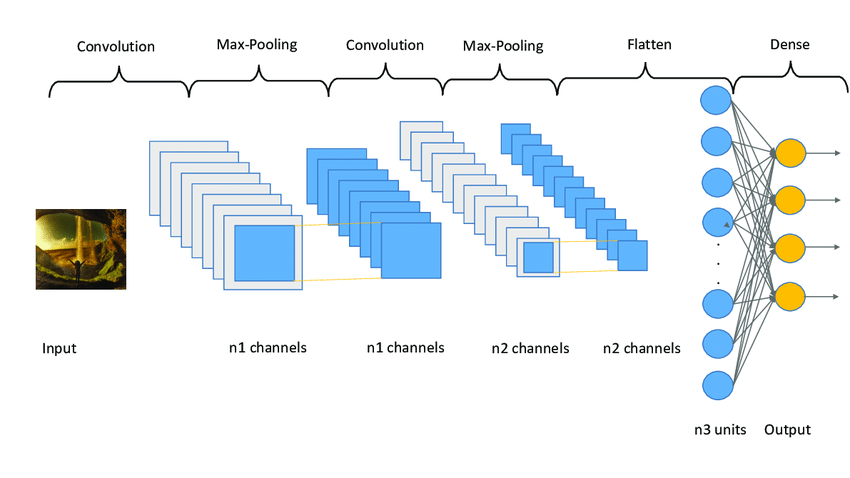
### Konvolúciós neurális hálók

(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 alábbi képen egy általános konvolúciós neurális háló felépítése látható a különböző rétegekkel.



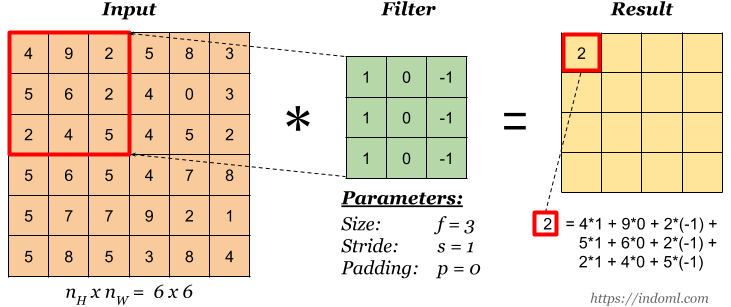
. ábra: Egyszerű konvolúciós neurális háló

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

A következő részekben a konvolúciós neurális hálók építőelemei kerülnek bemutatásra.

#### 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 a szakirodalom aktivációs térképnek (activation map) nevezi, amelyre utána további rétegek kerülhetnek, újabb konvolúciós réteg (convolutional layer), összevonó (pooling layer), vagy sorosító réteg (flatten layer).



. ábra: A konvolúció művelet bemutatása

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

#### Összevonó réteg (pooling layer)

Ennek a rétegnek az aktivációs térképek dimenziójának 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 csúsztatása, ez a stride. Azt adja meg, hogy hány sornyit/oszlopnyit mozdul el a filter a következő mintavételezéshez.



. ábra: average és max pooling rétegek példája

<https://tex.stackexchange.com/questions/654546/tables-side-by-side-with-big-curly-brackets>

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

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

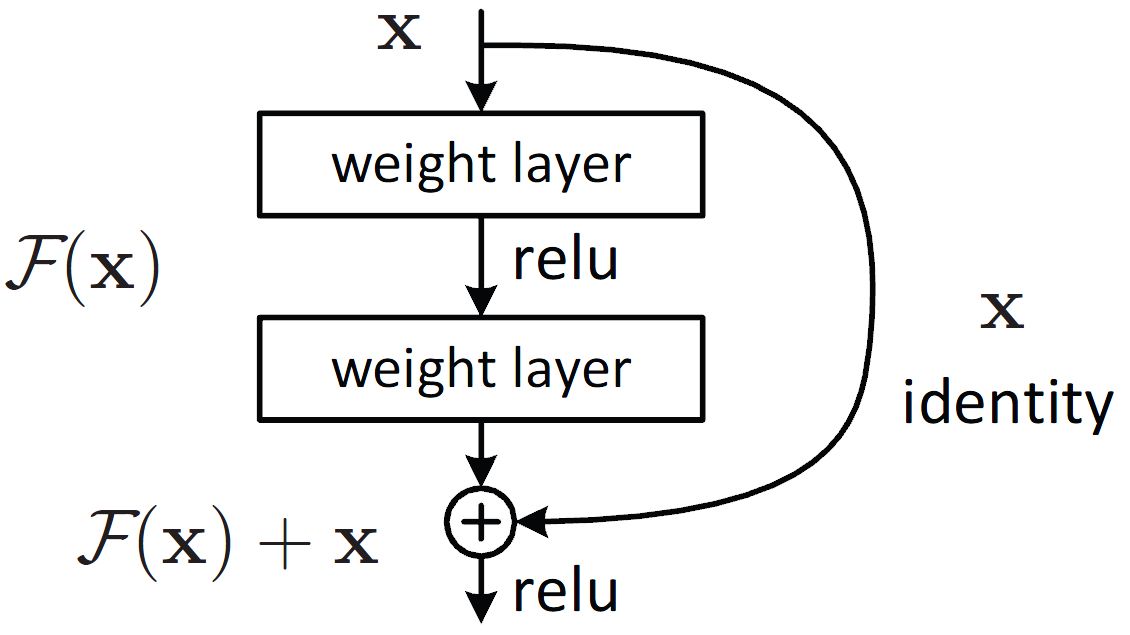
<https://paperswithcode.com/method/average-pooling>

<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. Az előrecsatolt (feedforward) neurális hálóknál a bemenet az összes rétegen keresztülmegy, egy útvonal van. A residual connectionnel ellátott hálók esetében számos útvonal van, amin a bemenet a háló kimenetéhez érhet. A több útvonal, azaz több részben különböző háló együttes eredménye lesz 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.

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

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

# 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, 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. Akár több millió kép is kellene tanítási halmazként egy ténylegesen hatékony és elfogadhatóan 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 adataugmentációval (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, felhasznált 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 a nyersképpel megegyező névvel. A Lightroom így amikor importálja a nyersképeket, rögtön alkalmazza a mellette található korrekciós fájlt is.

* 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. Ilyenkor a Lightroom belül lehet még alakítani a képeken, ha ez szükséges.

# Felhasznált technológiák, eszközök

Ebben a fejezetben az alkalmazáshoz felhasznált technológiákat ismertetem.

**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 célszerű volt ezt megtanulni. A pythonon kívül az R nyelvet szokták még használni, de egyre kevésbé elterjedt. Eddigi egyetemi kurzusokon mindenhol a python volt a választott nyelv, abban kellett feladatokat megoldani. Ezen felül a keretrendszerek, könyvtárak is, amiket használok python nyelven teljesek.

**Visual Studio Code**

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

**Google Colab, jupyter**

Mivel a neurális hálók tanítása nagyon számításigényes művelet, ezért a tanítást nem lokálisan végzem, hanem Google Colab-ban, egy jupyter notebookban. 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 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 fizetős csomagban egy adott mennyiségű számítási időt lehet vásárolni (~50 óra), ez nagyjából elégnek bizonyult.

**A GPU computation 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 mátrixműveleteket végezni.

CPU-n futtatva a tanítást egy 70 képes adathalmazon, nagyjából 8-9 másodperc volt egy tanulási ciklus (epoch) tanítási ideje, addig 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 esetemben 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 megvalósítanom a különböző neuronokat, rétegeket, hibafüggvényeket és még sok más, alacsony szintű kódot. A tensorflow a neurális háló GPU-n való futtatásáért felel.

**plotly**

A generált tanulási és jósággörbéket a plotly csomag segítségével rajzoltam ki, illetve tensorboard-ot 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 használtam numpy-t, ez szintén egy gyakran használt könyvtár 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.

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

# Az 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 nyersképek mellé
* opcionálisan Lightroom meghívása, eredmények megtekintése

### Adatok előfeldolgozása

A tanulási adathalmaz .NEF, .CR2, vagy .ARW fájlformátumú nyersképekből áll. Mindegyikhez fájlhoz 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. Mivel 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, fehéregyensúly, árnyalat, világosság, kontraszt, vibrálás 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.

A feldolgozás során első lépés volt az adatok megtisztítása. Érdemes manuálisan egy előválogatást tartani, ahol 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 nyer 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 értékeit [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 blokkokat és inception architektúrával) is [299x299x3] bemeneti dimenziókkal dolgozik. Opcionálisan mentésre kerül 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 választottam. Az .xmp-t beolvasva, az exiftool parseolja, majd a kiolvasott tageket kulcs-érték párok formájában visszaadja. 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"*), ezeket egy listába kerülnek eltárolásra, majd ezeket az elvárt kimenetek listájához kell hozzáfűzni.

<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. A két lista felcímkézése miatt a listák egy dictonarybe kerülnek, ez a későbbi feldolgozás során könnyít. Ez után egy *datasets.Dataset* osztályba rakom el, 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.

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

Az adatok előfeldolgozása lokálisan ennyi volt, az adathalmazon végzet további műveletek már a Colab-ben történnek. Természetesen ez a lokális előfeldolgozás is nagyon számításigényes, nagyjából a kisebb, 70 kép előfeldolgozása nagyjából 1ö 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 28 MB körül van.

### Neurális hálók implementálása

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

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 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 robusztusabb 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 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 tízes nagyságrendben található), ezért 5 külön neurális hálót tanítok be, tehát 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 átdimenzionálás a 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. Akkor még nem volt feltétlenül meg a mély hálózatok tanításához szükséges számítási kapacitás, amivel ki lehet váltani a nagyobb rétegeket sok kisebb réteggel.

<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 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.

<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, 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 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, azaz 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.

<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.

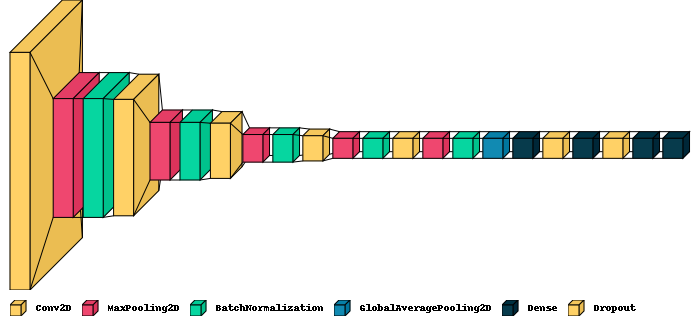
A modellépítő függvények üres Sequential (a Model-ből specializált osztály) vagy Model példányra kezdik el építeni a rétegeket és ilyen típussal is térnek vissza. Alapvetően Sequential-t és a keras szekvenciális API-ját használtam az egyszerűbb deklarálási mód miatt. Model-re akkor volt szükség, amikor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

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

### Modellek létrehozása

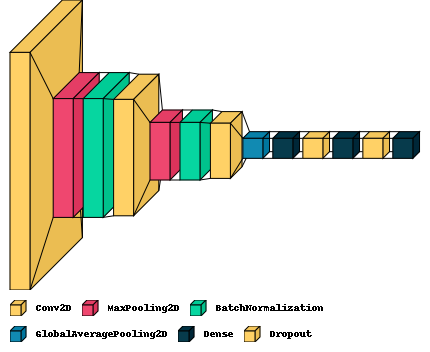
Mindegyik modellnek van egy saját modellépítő függvénye, ami létrehozza az adott hálózatot.

*buildNN\_init*: egy egyszerű mély konvolúciós hálót hoz létre. Felépítése:



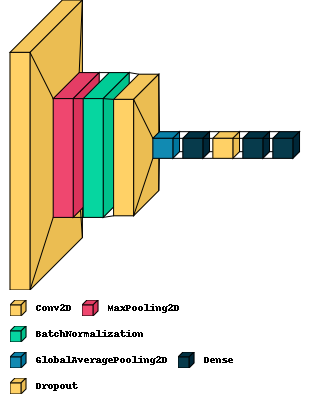
8. ábra: A *buildNN\_init* architektúra felépítése

*buildNN\_s*: Az előző háló egyszerűbb verziója, egyel kevesebb Conv-Relu-MaxPool-BatchNorm blokkal, kisebb filterméretekkel, valamint egyel kevesebb teljesen csatolt réteggel.



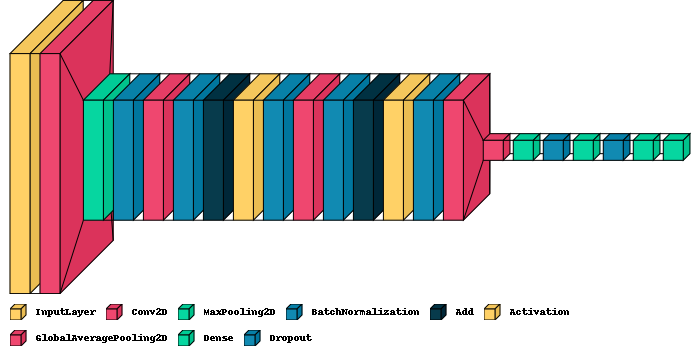
9. ábra: A *buildNN\_s* architektúra felépítése

buildNN\_xs: A *buildNN\_s*-hez képest kisebb háló, kevesebb rétegből áll.



10. ábra: A *buildNN\_xs* modell felépítése

*buildNN\_RCNN*: A *buildNN\_init*-ből indult, a konvolúciós rétegek között van egy visszacsatolt kapcsolat (residual connection).



11. ábra: A *buildNN\_RCNN* architektúrája

*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. A MobileNetV2 és InceptionV3 hálókkal kísérleteztem alaphálónak véve, nem bizonyultak elég hatékony megoldásnak, túl komplex volt az alapmodell.



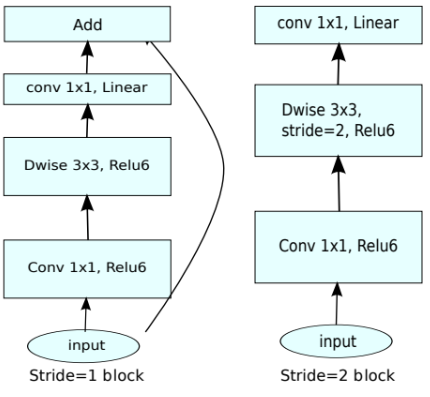
12. ábra: a *buildNN\_pretrained* architektúrája

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

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



13. ábra: Az InceptionV3 architektúrája



. ábra: A MobileNetV2 architektúrája

<https://paperswithcode.com/method/mobilenetv2>

### 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 hálózat, akkor az alaphálót taníthatóvá 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.

### 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:

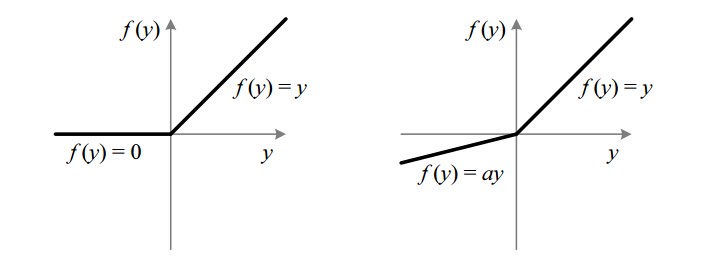
* neuronok száma (hány neuron van egy rétegben, mekkora a konvolúciós, összevonó filter)
* rétegek száma (hány réteg van egymás után, hány konvolúciós filtert teszünk a hálóba)
* dropout alkalmazása (van-e a hálóban dropout réteg és ha van akkor a kapcsolatok hányadrésze lesz véletlenszerűen elhagyva)
* 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 rész a kipróbált és felhasznált aktivációs függvényekkel (activation functions), hibafüggvényekkel (loss functions) és optimalizálókkal (optimizers) foglalkozik.

#### 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 aktivációs függvények képe

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 nagyon nagy, illetve nagyon kicsiny *x* értékek esetében. A gyakorlatban a legtöbb regressziós problémánál a ReLU, vagy valamilyen változatát alkalmazzák gyors számíthatósága és egyszerűsége miatt.



16. ábra: A sigmoid függvény képe

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

#### 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, azon felül MAE hibafüggvény.
* Mean Square Logarithmic Error (MSLE): Az elvárt és prediktált érték hányadosának logaritmusát kell venni, majd az összes mintára számtani közepet számítani.

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

#### Optimalizálók (Optimizers)

Az optimalizálók olyan algoritmusok, amik a súlyok frissítésének helyes mértékéért felelnek, azaz a háló konvergenciájáért felel. A háló tanítása során az a cél, hogy a súlyok úgy frissüljenek, hogy a hibafüggvény értéke csökkenjen és a hiba minimalizálódjon. A hibafüggvény megadja a háló hibáját, ezt 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 szokták megszorozni. Az optimalizáló algoritmusok ezt a tanulási rátát vagy rátákat módosítják, adott esetben adaptívan a gradiens nagyságához, vagy az adott 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.

<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>

A következő részben a modellek teljesítményének elemzése és a modellek finomhangolásáról lesz szó.

# 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: ezek 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.

### 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 az vehető észre, hogy a modell nem képes tovább javulni és alacsonyabb hibával prediktálni.

<https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>

### Adathalmazok

A tanuláshoz felhasznált adatok:

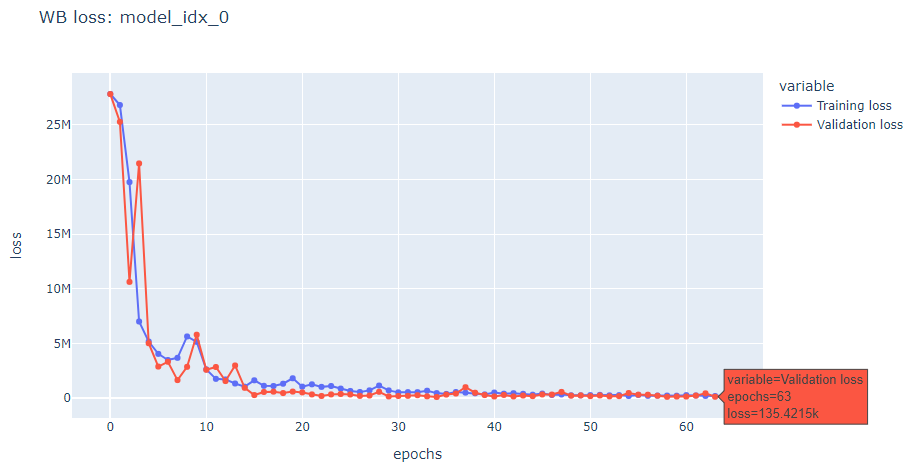
* Konstanz adathalmaz: Az adathalmaz egy nyári, konstanzi kirándulás képeiből áll, összesen 56 kép a tanulási halmaz, de ez a duplájára nőtt az augmentáció miatt. A validációs halmazban 12 kép található, ugyan ebből az adathalmazból. Jellemzően nappali, jó megvilágítású képek.
* Szakestély adathalmaz: 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.

### Modellválasztás

A modellválasztást a fent definiált 5 architektúra vizsgálatával kezdtem egy alap paraméterezéssel a Konstanz adathalmazon. Minden háló 200 tanulási ciklust hajt végre, ha early stopping miatt meg nem áll hamarabb. 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 legutóbbi legjobbnál, akkor leáll a tanulás, illetve 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 mind az 5 háló egyetlen kimeneti paraméterre, a fehéregyensúlyra tanul, így segítve a különböző modellek közötti teljesítménybeli különbségek észrevételét és kiértékelését.

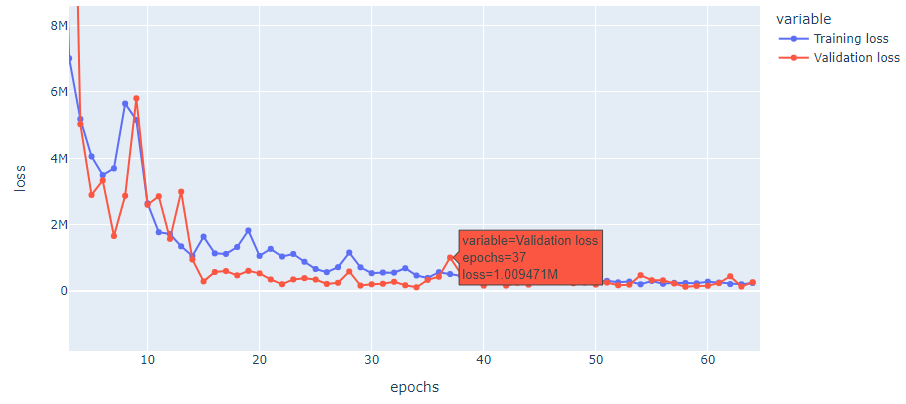
#### Szekvenciális konvolúciós modellek

Az alábbi 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 batch-enként 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.



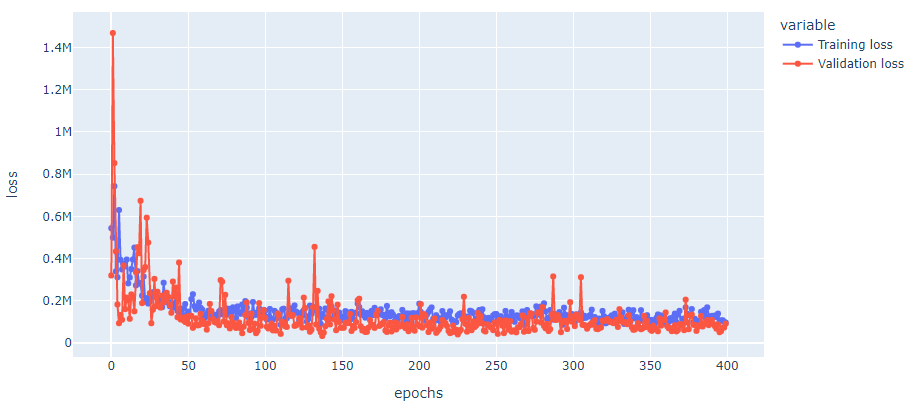
. ábra: A *buildNN\_init* háló tanulási görbéje

A fenti 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.

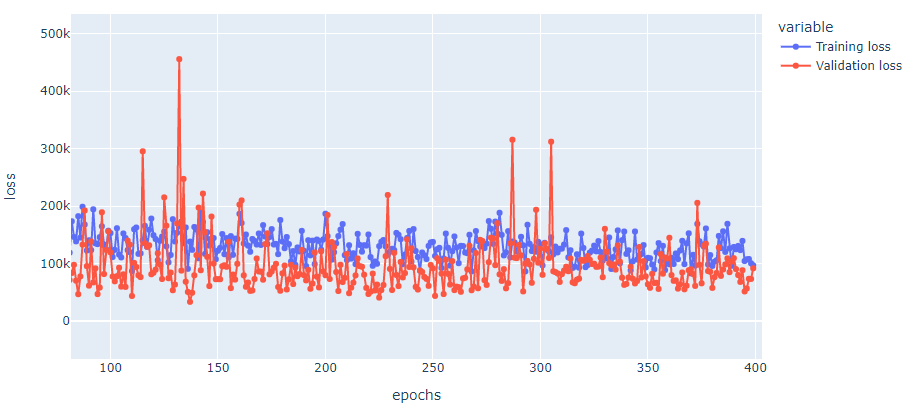


. ábra: A *buildNN\_init* háló tanulásának nagyított grafikonja

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



19. ábra: A *buildNN\_init* háló tanulása early stopping nélkül



. ábra: A *buildNN\_init* háló hibagrafikonja nagyítva, early stopping nélkül

A fenti 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. Szerencsére az optimalizáló és a hibafüggvény is jónak tűnik, így ezeken nem szükséges változtatni.

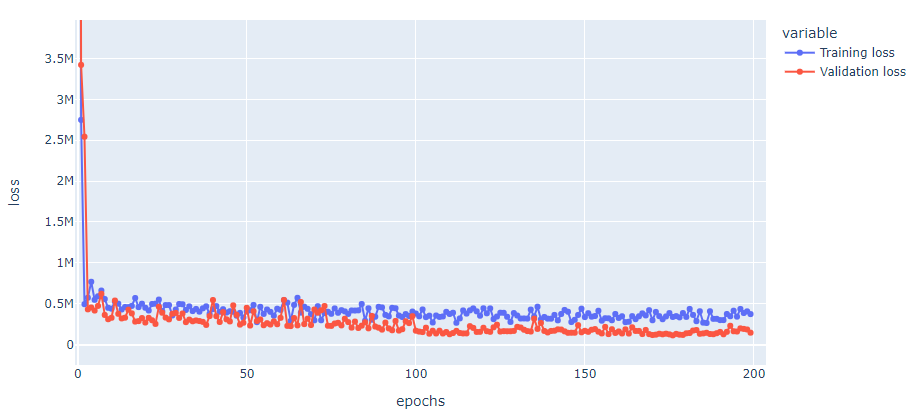
<https://analyticsindiamag.com/what-is-the-plateau-problem-in-neural-networks-and-how-to-fix-it/>

<https://stats.stackexchange.com/questions/523281/is-it-normal-for-training-loss-to-plateau-before-decreasing>

Important:

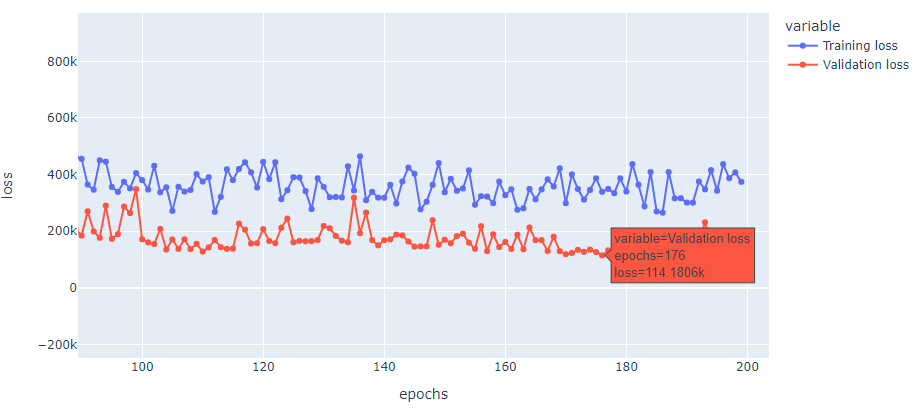
<https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>

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:



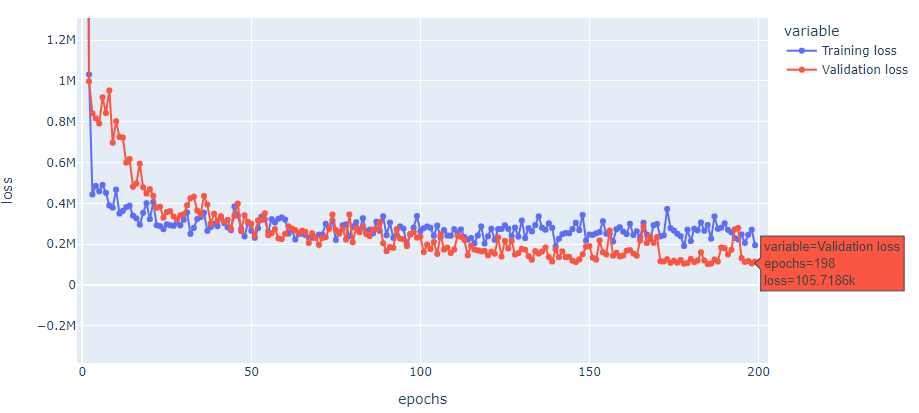
. ábra: a *buildNN\_s* modell tanulási görbéje

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



. ábra: A *buildNN\_s* háló tanulási görbéje, 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 hibaértékekre képes, mint az egy szinttel komplexebb háló, a *buildNN\_s.*



. ábra: A *buildNN\_xs* tanulási görbéje

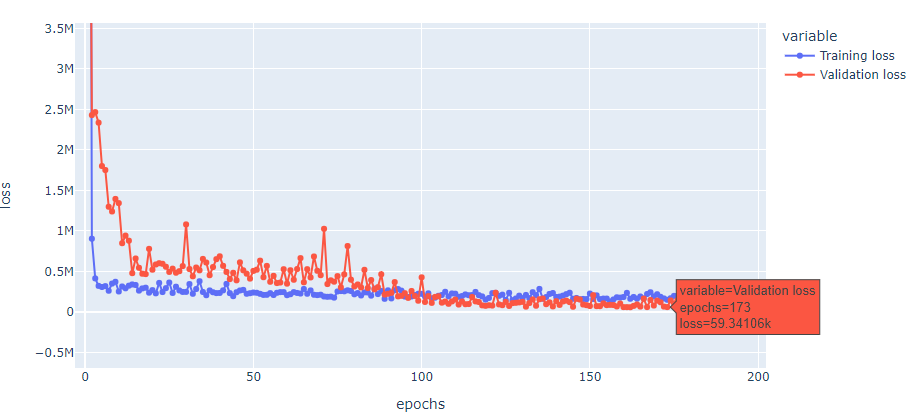
#### Visszacsatolt háló

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.

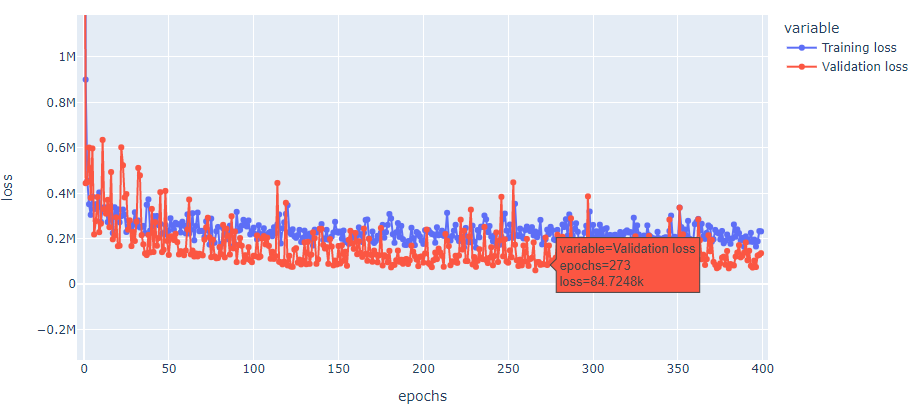
A visszacsatolt kapcsolatok (residual connection) miatt több neurális hálót szimulál egy architektúra, mivel a gradiensek frissítésének több útja van, így a háttérben nagyjából úgy viselkedik, mint több háló.

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



. 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.



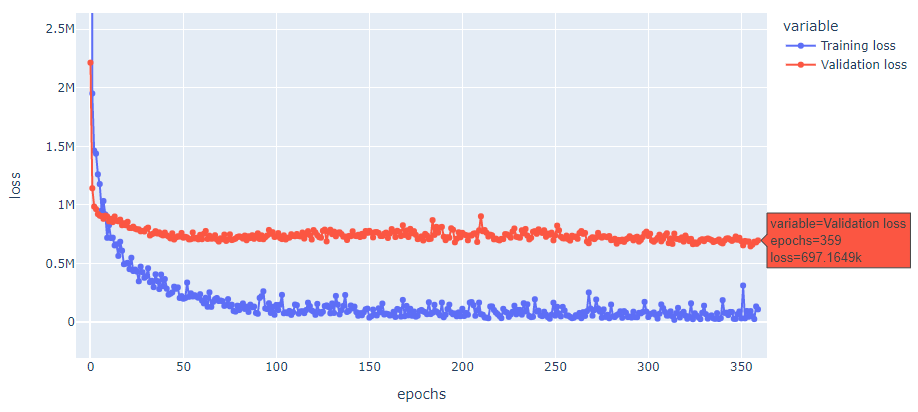
. ábra: A *buildNN\_RCNN* modell tovább tanítva. Az eredmény nem lett jobb.

#### Transfer learning

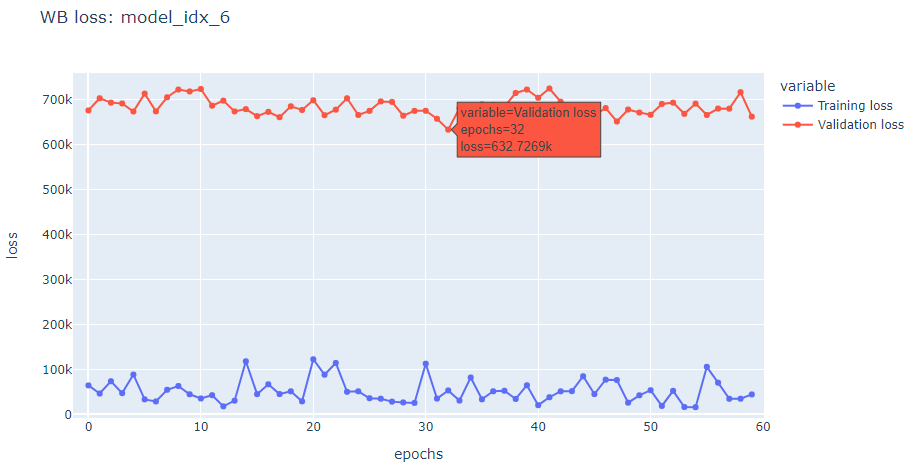
A transfer learning, mint módszer leírása korábban, az [5.7 Transfer learning](#_Transfer_learning) részben található. Ebben a részben a transfer learning alkalmazásáról van szó.

* + - 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.



. ábra: Előtanulási fázis az InceptionV3 hálón



. ábra: Utótanulási fázis az InceptionV3-on, a *buildNN\_xs* teljesen csatolt (fully connected) rétegeivel

A fentebbi 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\_init* 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 az alábbi táblában látható:

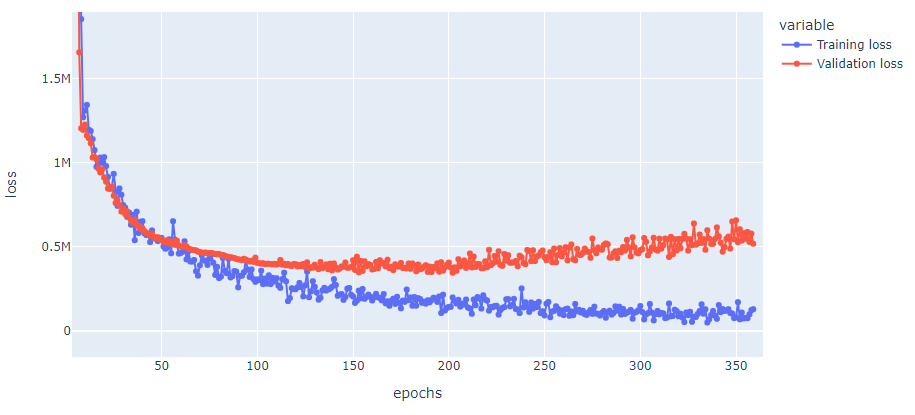
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Model | Size (MB) | Top1-Acc | Top5-Acc | #params | depth | time/inference (ms, on CPU) | time/inference (ms, on GPU) |
| InceptionV3 | 22 | 77.9% | 93.7% | 23.9M | 189 | 42.2 | 6.9 |
| MobileNetV2 | 14 | 71.3% | 90.1% | 3.5M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

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

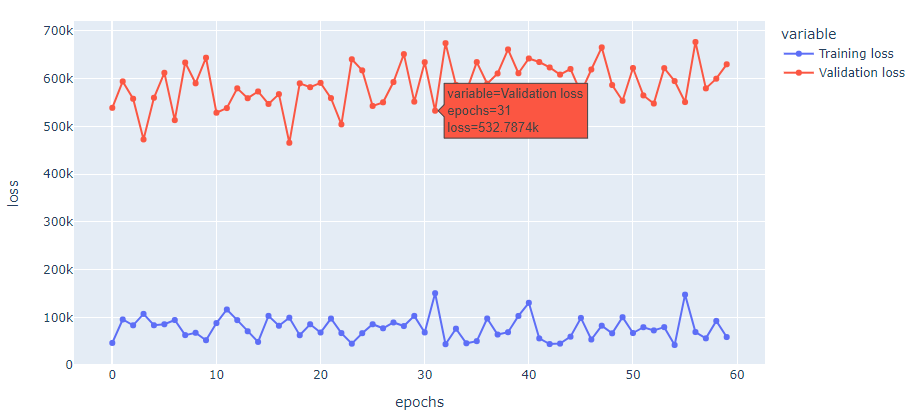
A fenti táblázatban a paraméterek száma a teljes modellre vonatkoznak. A korábbi, InceptionV3-ra épített háló csak 22 millió paraméterrel rendelkezett a teljesen csatolt rétegek neuronjainak alacsony száma miatt, míg itt ~24 millió paramétert ír a táblázat.

* + - 1. 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 korábbi hálókhoz képest magas hiba mellett (~400.000). Így a háló finomhangolásakor az utótanulási fázisban nem volt semmiféle elvárásom a javulásra, ez be is igazolódott.



. ábra: A *modelNN\_pretrained* háló előtanulási görbéje a MobileNetV2 alapokon

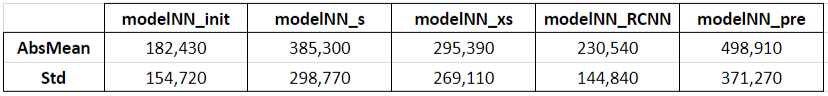


. ábra: A *modelNN\_pretrained* háló utótanulási görbéje a MobileNetV2 alapjain

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.

#### 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.



. ábra: Az eddig tárgyalt modellek hibastatisztikái

Az hibákat összevetve a *buildNN\_init*, vagy a *buildNN\_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.

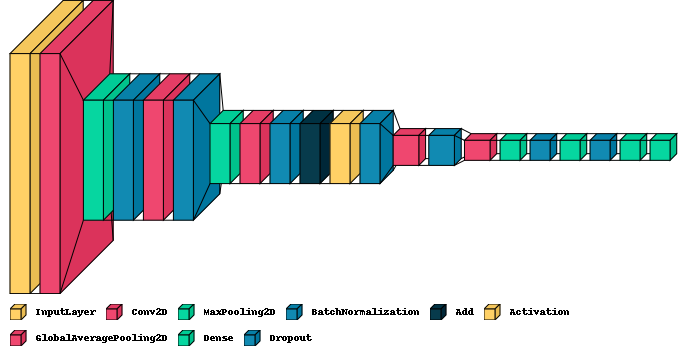
Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. ábra: Hálók hibáinak részletes összehasonlítása a validációs halmazon

#### Még egy visszacsatolt konvolúciós háló

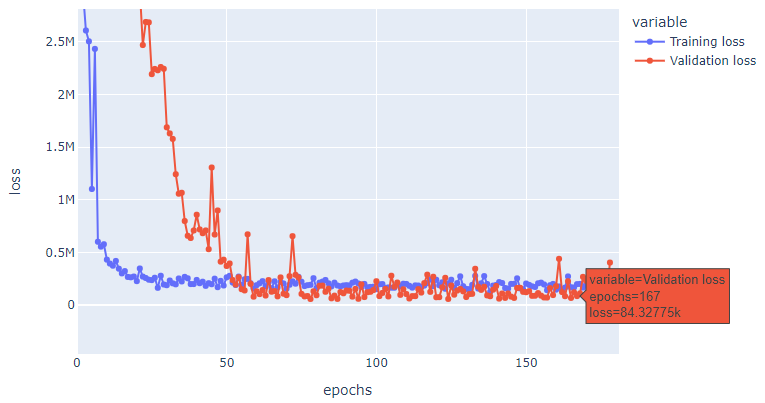
A fenti statisztika alapján érdemesnek találtam a *modelNN\_init*, és a *modelNN\_RCNN* hálók tulajdonságait egyesíteni, hiszen ezek adták a legjobb hibaértékeket. Az új háló a *modelNN\_irv2* lett.



. 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



. ábra: A *modelNN\_irv2* modell tanulási görbéje

Jelentős minőségi javulást sajnos nem lehet észre venni, az elődmodellekhez hasonlóan 80-100 ezres hibák fordulnak elő, mint a két legjobb elődhálónál. 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 milliós nagyságrendbe kellene elérhető és jó tanulási képadatoknak lennie a prediktálni kívánt tulajdonságokkal együtt, sajnos ilyen adathalmaz jelenleg nincs. Ezzel a problémával viszont a többi modellnek is meg kell küzdeni.

#### 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 az early stopping segítségével kivédhető.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. 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 a másik jól teljesítő modellel (*modelNN\_irv2)*.

#### Különböző modellparaméterezések

Általánoságban a MSE (mean square error) jó a regressziós problémákhoz, ugyanakkor érdemesnek gondoltam 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 legtöbb modellhez. Ennek ellenére kipróbáltam még az Adagrad és az RMSProp optimalizálókat. 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.

* + - 1. Hibafüggvények:

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

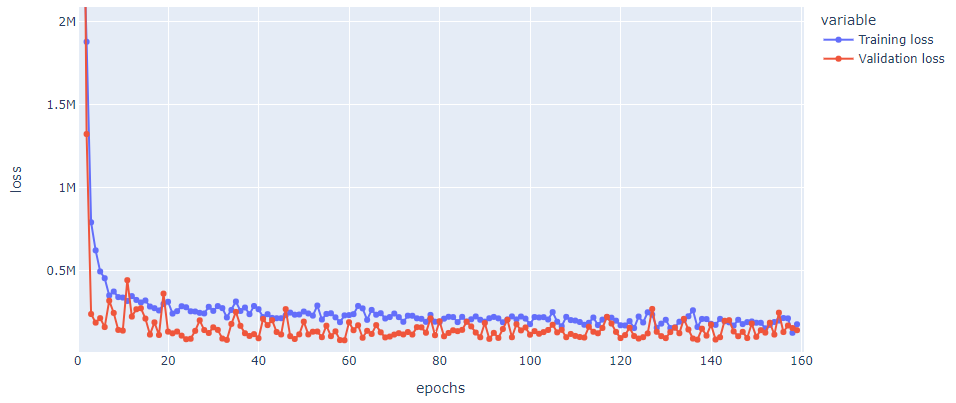
. ábra: A különböző hibafüggvények hálókra gyakorolt hatása

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 teljesítményén.

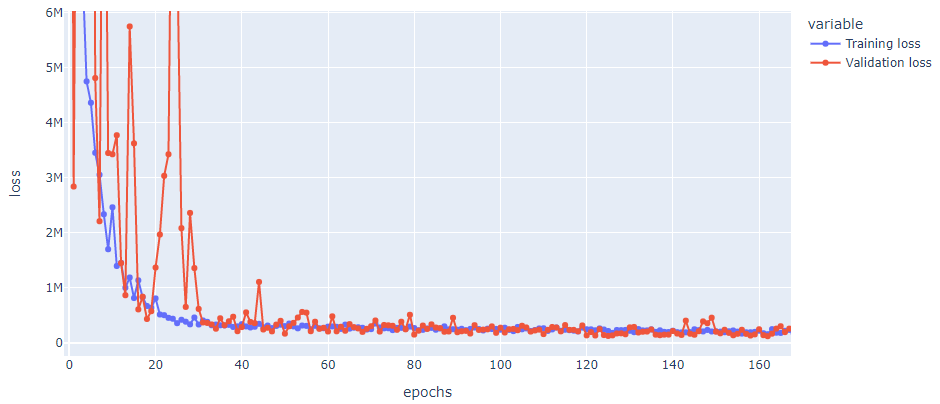
A továbbiakban a modelNN\_irv2 hibafüggvénye Huber loss lesz, a modelNN\_init hibafüggvénye marad MSE (mean square error).

* + - 1. Optimalizálók

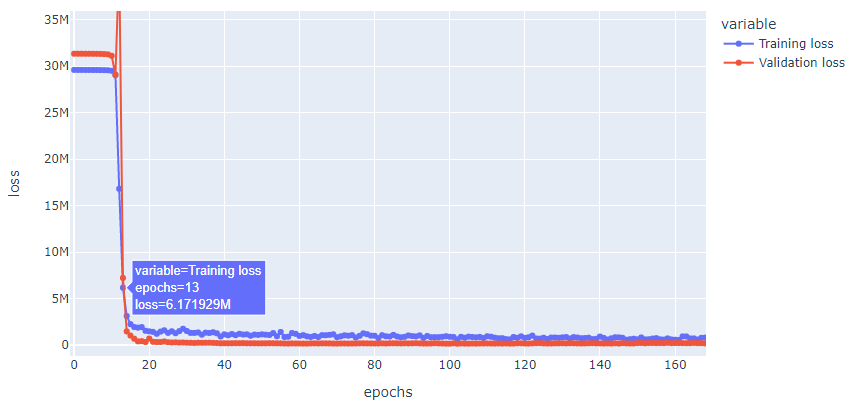
Az optimalizálók a lépésenkénti gradiensfrissítés mértékét befolyásolják, így a konvergencia sebességét is, ezért itt a tanulási görbéket érdemes kiértékelni. Az lent található tanulási görbék a *modelNN\_init* és a *modelNN\_irv2* hálók egy tanítását mutatják be Adam, RMSProp és az Adadelta optimalizálókkal.



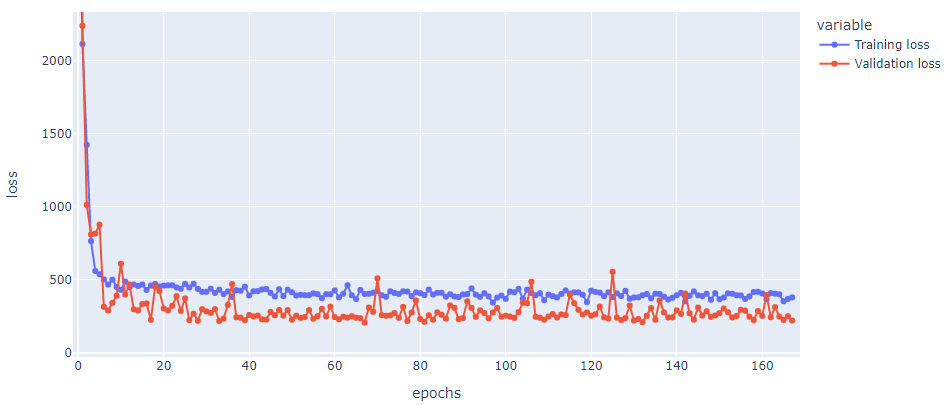
. ábra: *modelNN\_init* háló tanulási görbéje, Adam optimizer



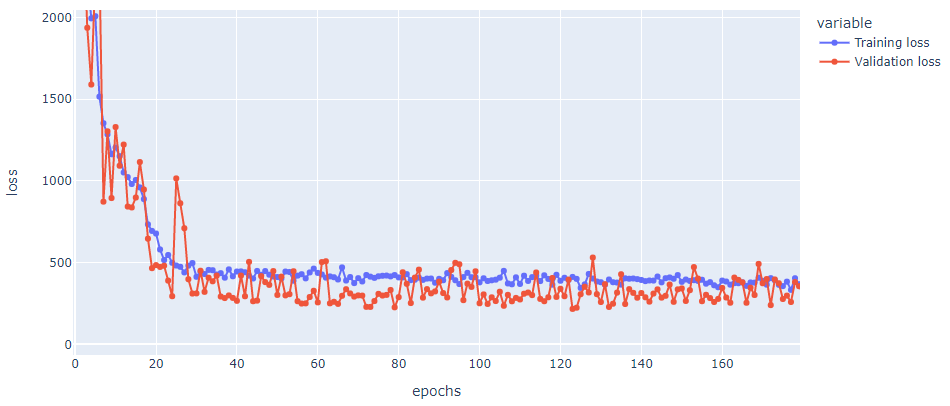
37. ábra: *modelNN\_init* háló tanulási görbéje, RMSProp optimizer



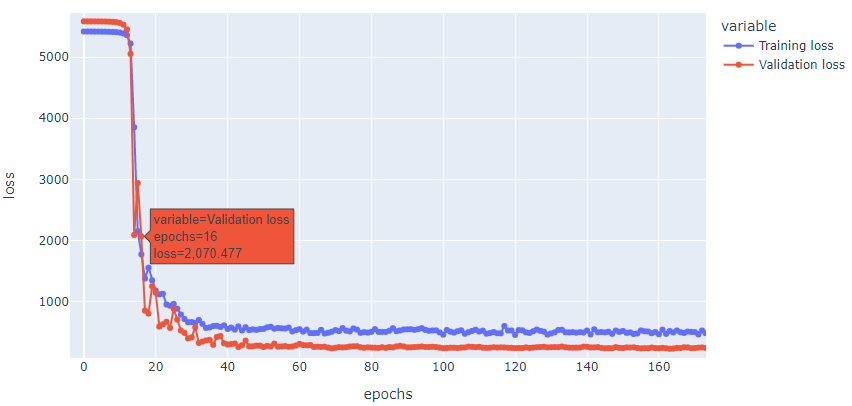
38. ábra: *modelNN\_init* háló tanulási görbéje, Adadelta optimizer



39. ábra: *modelNN\_irv2* háló tanulási görbéje, Adam optimizer



40. ábra: *modelNN\_irv2* háló tanulási görbéje, RMSProp optimizer



. ábra: *modelNN\_irv2* háló tanulási görbéje, Adadelta optimizer

Összehasonlítva a fentebbi tanításokat, 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). 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.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. ábra: Optimalizálók összehasonlítása a validációs hibákat tekintve. A *modelNN\_irv2* mindenhol megveri a *modelNN\_init*-et.

* + - 1. 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 érdemesnek a kipróbálásra. 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.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. ábra: A *modelNN\_init* és a *modelNN\_irv2* validációs halmazon mért hibái különböző batch méretek mellett (zöld: legjobb érték, sárgás: második legjobb érték)

#### 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.

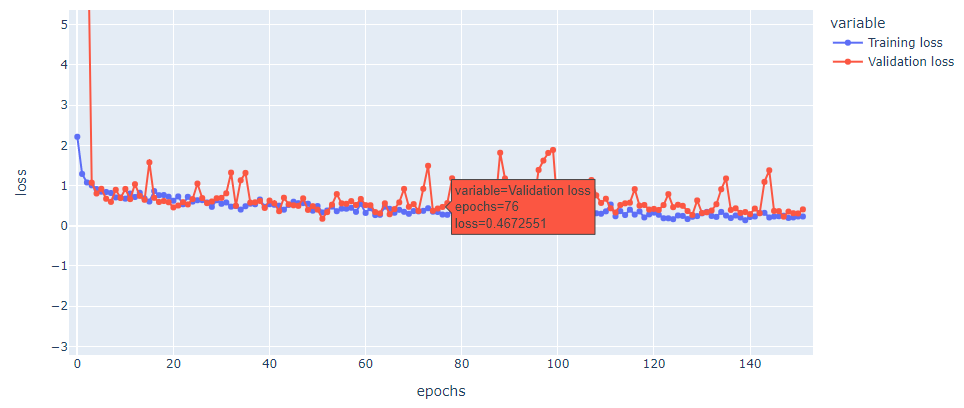
A korábbi tapasztalatok alapján a *modelNN\_irv2* háló jó lesz a fehéregyensúly prediktálására.

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.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

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



. 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 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.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. 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 intervallumba esik, de nem szoktak nagyon a tartományon kívül értékek előfordulni.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. 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 intervallum, a gyakorlatban nem igazán fordulnak elő az intervallumon kívüli értékek.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. ábra: A modellek hibái vibrálás esetén

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

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

. ábra: Választott modellek és főbb paramétereik a tulajdonságok prediktálásához

### Betanult modellek mentése, betöltése

A tanult modellek HDF5 fájlokként (.h5 kiterjesztés), opcionálisan dátummal ellátva mentésre kerülnek a 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.

<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. 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.

### Jellemzők prediktálása

A projekt ezen része az, ami a felhasználók számára készült, 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.

Alább látható 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

#### 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 eredményben ezen kívül az adathalmazok végessége, illetve a kis méret is korlátot szab.

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

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

<https://www.cambridgeincolour.com/tutorials/white-balance.htm>

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.

|  |  |  |
| --- | --- | --- |
| Nyerskép | Kézi retusált | Neurális háló retusált |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Munka értékelése, továbbfejlesztési lehetőségek

Szakdolgozat során számos új problémával kellett szembesülni és megoldani, amik korábbi tanulmányaim alatt, az elméleti alapok elsajátításakor nem merültek fel problémaként. Ilyen volt többek között az adatok előfeldolgozása és a kis adatmennyiség, sajnos ezek miatt talán 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, ilyen feltétel 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 rengeteg 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, megoldási ötleteim, amiken dolgoztam a szakdolgozat alatt.

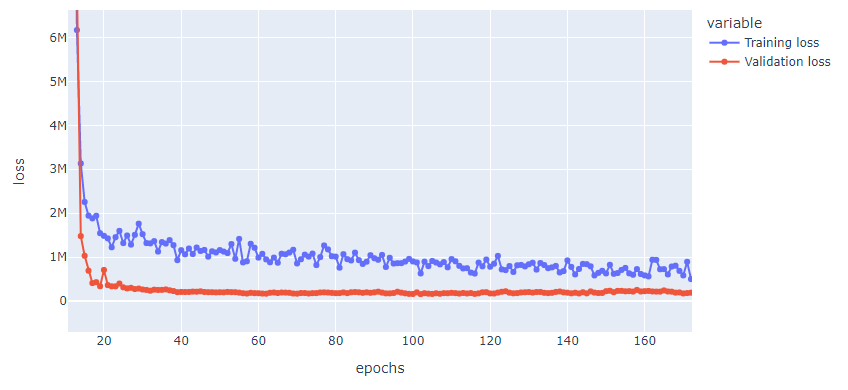
Szeretném megköszönni két Simonyis ismerősömnek, Bogár Richárdnak és Kurgyis Pálnak a hasznos és tartalmas beszélgetéseket a neurális hálókkal kapcsolatban, ezek igen sokat segítettek.

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.

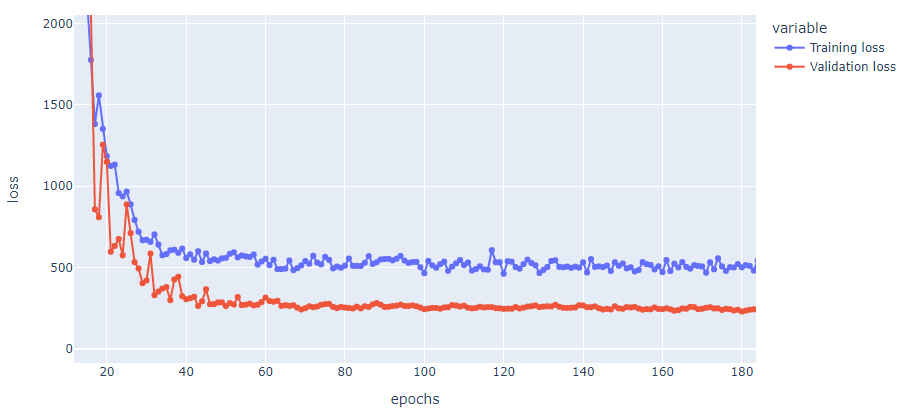
# Irodalomjegyzék

1. Google. (2022. November Scholar). *Google Scholar*. Forrás: Google Scholar: https://scholar.google.com/citations?view\_op=top\_venues&hl=en&vq=eng
2. *Wikipedia - Claude Shannon.* (2022). Forrás: https://en.wikipedia.org/wiki/Claude\_Shannon
3. *Wikipedia - John McCarthy*. (2022). Forrás: https://en.wikipedia.org/wiki/John\_McCarthy\_(computer\_scientist)
4. *Wikipedia - Marvin Minsky*. (2022). Forrás: https://en.wikipedia.org/wiki/Marvin\_Minsky

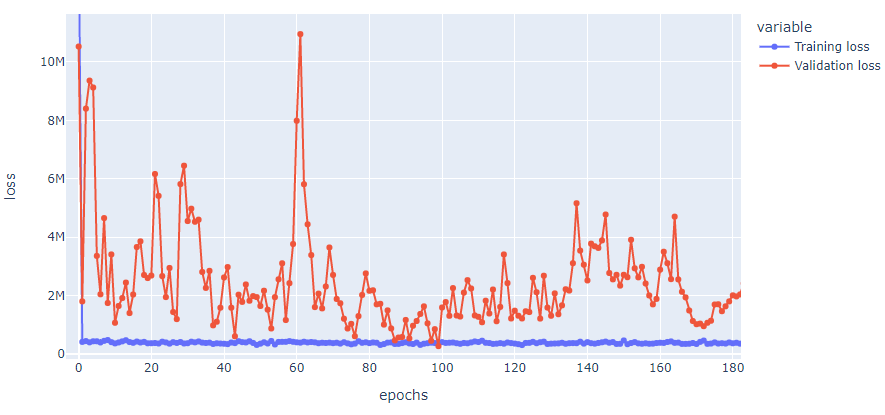
# Függelék



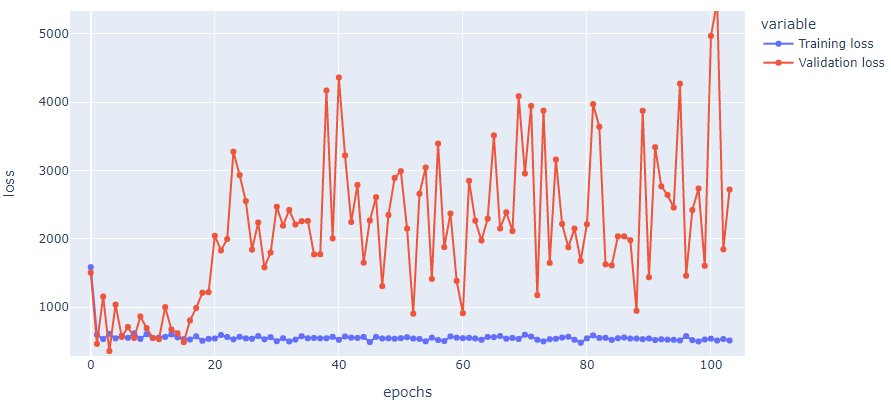
. ábra: A *modelNN\_init* háló Adadelta optimalizálóval, nagyított hibagörbe



. ábra: A *modelNN\_irv2* háló Adadelta optimalizálóval, nagyított hibagörbe



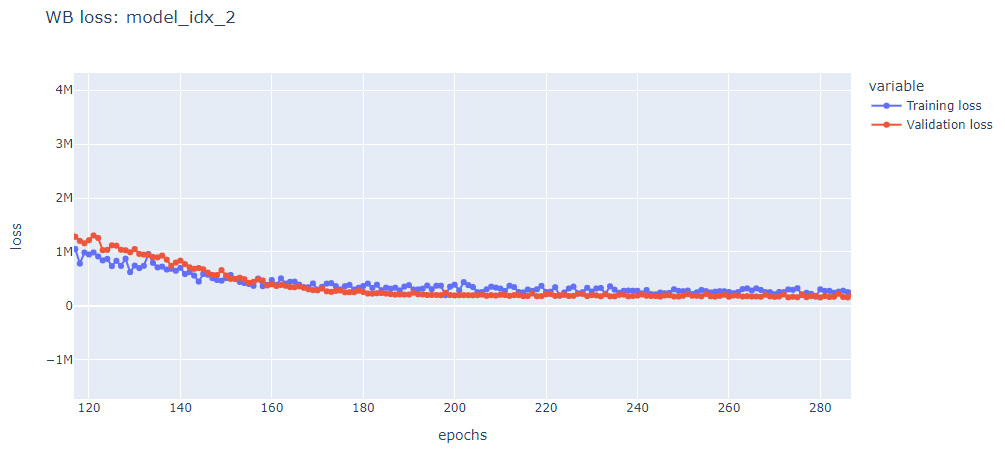
. ábra: A *modelNN\_init* háló tanulási görbéje, egyes batch mérettel. Láthatóan óriási kilengések vannak a validációs halmaz hibagörbéjében.



. ábra: A *modelNN\_irv2* háló tanulási görbéje, egyes batch mérettel. Láthatóan óriási kilengések vannak a validációs halmaz hibagörbéjében.

TODO place other graphics and stats here?

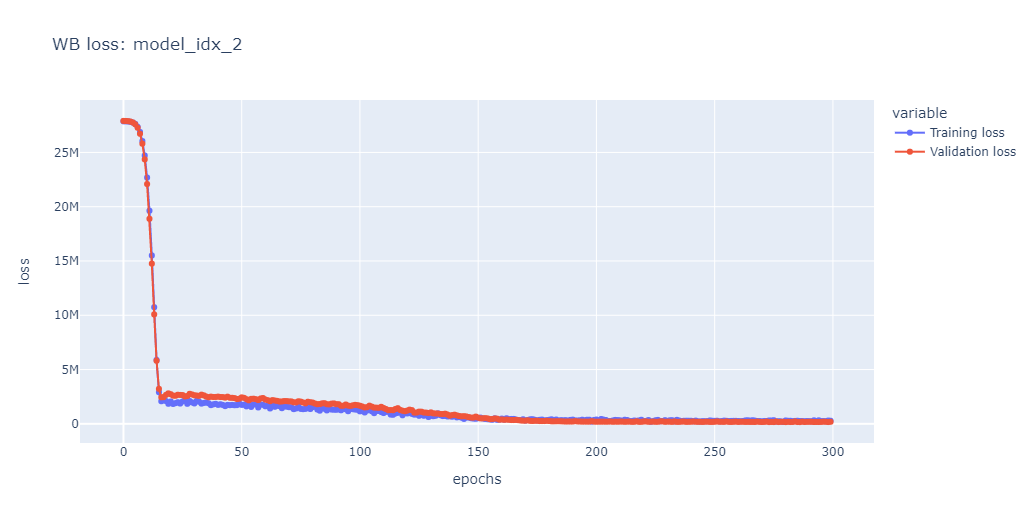
Itt érdemes megvizsgálni a 160. tanítási ciklus környékét:

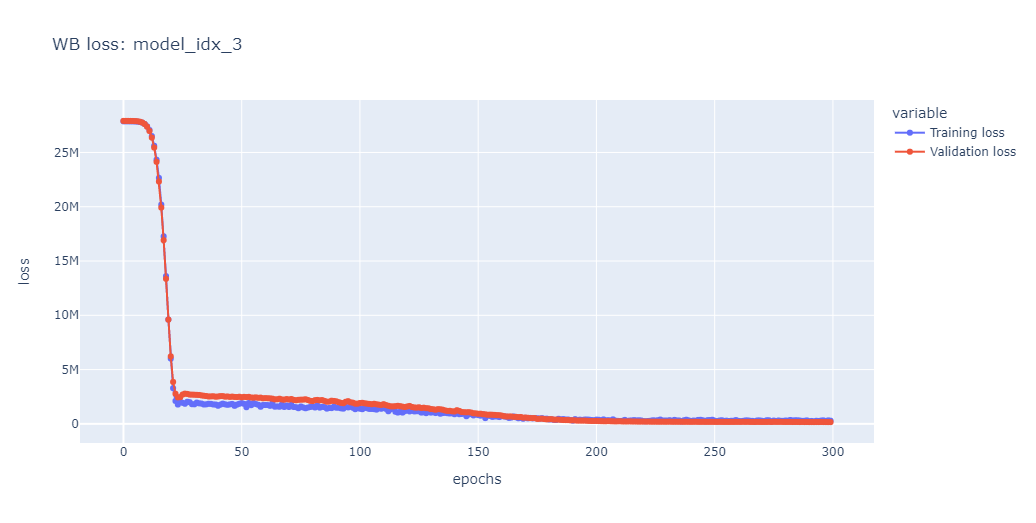


46. ábra: A 160. epoch és környezete a *buildNN\_xs* háló tanulási görbéjén

A grafikonon az látható, hogy nagyjából a 160. tanítási ciklus után jobb eredményt ér el a háló a validációs adatokon, mint a tanulási adatokon és stabilan 180.000 körül mozog a hiba egy batchre nézve. Ez jobb, mint az előző két háló eredménye. Problémát ilyenkor a validációs adatok jósága jelenthet, ha a validációs adatokra kezdettől fogva jobb értékeket prediktál a háló, mint a tanulási adatokra. Ha ez lenne a helyzet, akkor a validációs adathalmaz nem elég reprezentatív, azaz nem hordozzák a validációs adatok ugyan azt a komplexitást, mint a tanulási adatok. Jelen esetben nem ez a helyzet, mivel a validációs halmaz a tanulás kezdetén magasabb hibát produkált, mint a tanulási halmaz.

A két háló tanulási görbéje is igen hasonló, mivel a két háló is csak 2 réteg konfigurációjában tér el egymástól. A tanulási görbék elején figyelhető meg egy nagyon kis eltérés a tanulási sebességben. A komplexebb háló (első grafikon – *model\_idx\_2*) gyorsabban tanul a kezdeti fázisokban, mint az egyszerűbb (második grafikon – *model\_idx\_3*)





47. ábra: A *buildNN\_xs* és *buildNN\_xxs* hálók tanulási görbéi. A *buildNN\_xs* kezdeti tanulása gyorsabb.

A tanulási ciklusok 20 ciklusonkénti mintavételezéséban a hibák összehasonlítása a két hálón:

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

48. ábra: a *buildNN\_xs* és a *buildNN\_xxs* által létrehozott hálók hibaértékeinek összehasonlítása

Jelen esetben a nagyobb háló jobbnak látszik, hiszen a validation loss alacsonyabb. Ugyanakkor ez nem egy jelentős a különbség, így mindkét hálót érdemes a későbbiekben tovább vizsgálni, illetve fejleszteni.

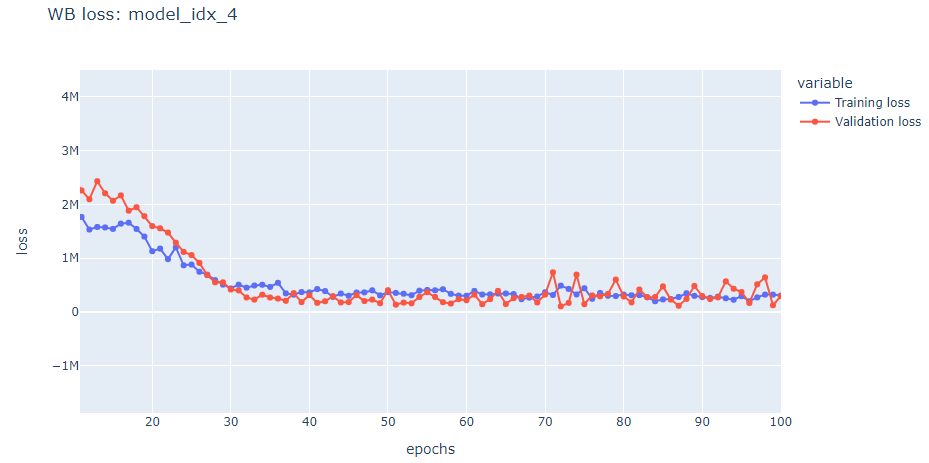
Az alábbi táblázatban a két háló validációs halmazon mért hibái láthatók elemenként. Ezen kívül a hibák abszolút értékének számtani közepét (mean), illetve szórását (std: standard deviation) is feltüntettem. Egy modell akkor működik jól, ha a hibák számtani közepe és a szórása együttesen kicsi.

Ein Bild, das Tisch enthält.

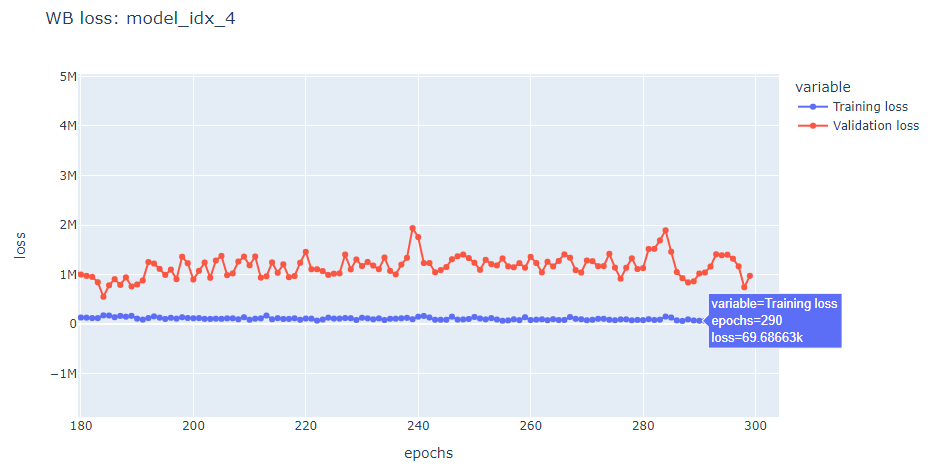
Automatisch generierte Beschreibung

49. ábra: A *modelNN\_xs* és *modelNN\_xxs* hálók hibáinak összehasonlítása a validációs hamazon

Az 50. epoch környékén a validációs hiba egy kis időre a tanulási hiba alá esett a modell, majd elkezd egyre változóbb és magasabb értékeket adni, míg végül a 150. epochtól teljesen elszakad a tanulási hibától. Ebből arra következtethetünk, hogy a modell túltanul, azaz overfitel. Ennek megoldása lehet a rövidebb tanulási idő vagy a kevésbé komplex modell.



50. ábra: Az 50. epoch környékén a validációs hiba alacsonyabb, mint a tanulási hiba



51. ábra: A validációs hiba értéke egyre emelkedik, a tanulási hiba tovább javul: tipikus overfitting

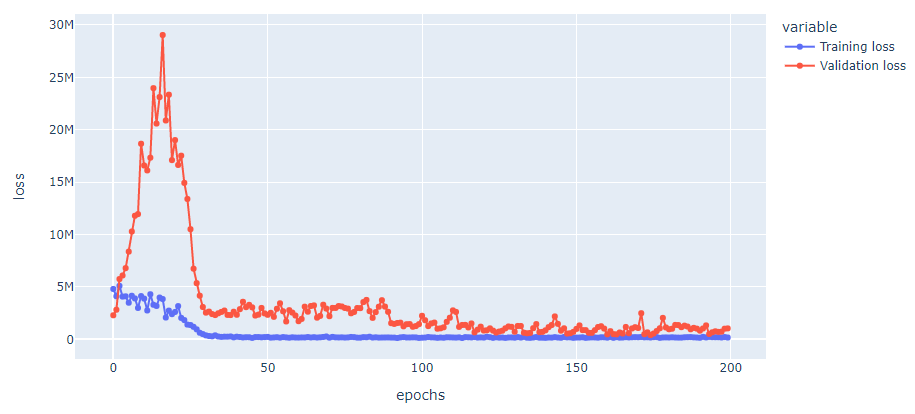
*buildNN\_xxs*: A *buildNN\_xs*-hez képest a változás, hogy kevesebb neuron alkot egy réteget.

Ein Bild, das Text, ClipArt, Vektorgrafiken enthält.

Automatisch generierte Beschreibung

52. ábra: A *buildNN\_xxs* modell felépítése

A negyedik hálóvariáns a *buildNN\_xxs* függvény eredménye, ez a háló kisebb konvolúciós filterekkel rendelkezik, mint a buildNN\_xs hálója, minden másban megegyezik a kettő.



Ez a háló nem tűnik alkalmasnak a célra, az alultanulás figyelhető meg rajta. Nem sikerül az adatok komplexitását jól reprezentálni, a validációs hiba ugyan megközelíti, de nem süllyed a tanulási hiba szintjéig, vagy alá. A modell nem elég komplex.