Témalabor dokumentáció

Kezdő lépések és félévi összefoglaló.

Kezdetek

Az elején nagyon sok új dolgot kellett megismerni, ezelőtt a csapat jeletős része még nem foglalkozott a Machine Learning témakörrel. És mivel ebben a témakörben nagyon elterjedt a Python nyelv használata először ezzel kellett megismerkedni mielőtt belevághattunk volna a munkába.

Python

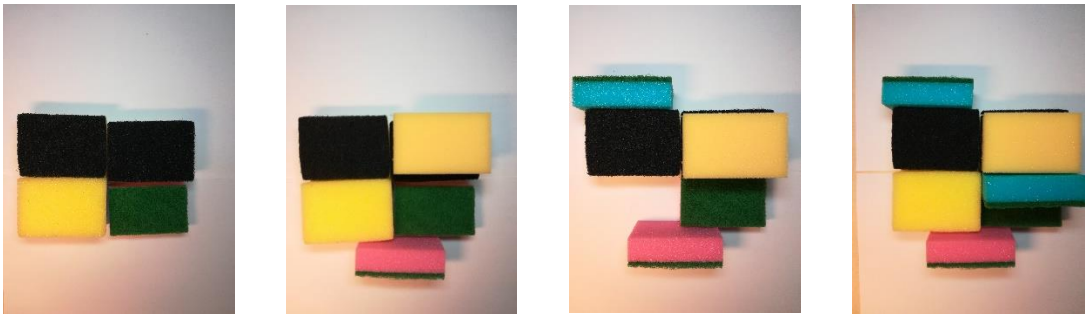
Először az volt a feladatunk, hogy ismerkedjünk meg a Python nyelvvel.

Kaptunk egy nagyon jó összefoglaló videót a Python nyelvről, amiből Hála Istennek sikerült megismerkedni a nyelv egyedi stílusával.

Gyakorlásképp más tárgyak házi feladatait is ebben a nyelvben írtuk. Pythonhoz Jupyter notebookot, Visual Studiot és PyCharm-ot használtunk. Jupyter notebook nagyon könnyen kezelhető és szerintem nagyon hatékonyan használható, ha a kódunk még kezelhető méretű, illetve komplexitású. Utána inkább Visual Studioban és PyCharmban érdemes folytatni a programozást, amiben nagyon hatékonyan lehetett fejleszteni és elég jól lehet debugolni.

Ezután már elkezdhettük a Machine Learning alapjait.

Szivacsfázisok megkülönböztetése



A téma kezdeti fázisában még nem tudtuk, hogy pontosan mit kell majd felismernie a programnak, ezért azt találtuk ki, hogy építünk magunknak saját modellt ameddig az igaziról nem kapunk képeket. Először Legóból akartunk építkezni, de mivel még azon a héten nem állt rendelkezésünkre szivacsból építettünk magunknak különböző fázisokat, amin tudunk gyakorolni. Meglepően jó eredmények jöttek ki még a legegyszerűbb modellek használatánál is, de igazából ez várható is volt a képek nagyon különböznek egymástól sok az eltérés könnyű a megkülönböztetés. Ennél a résznél kellett megismerkednem azzal, hogy hogyan lehet beolvasni a képeket mi a különbség a színes és a fekete fehér beolvasás között és hogy hogyan kell átalakítani a beolvasott képek tömbjét, hogy azt utána a tanítás után fel tudjuk használni.

Képek beolvasása

```
In [16]: DATADIR = "E:\Egyetem\Data\szivacs"
         CATEGORIES = ['phase_1','phase_2','phase_3', 'phase_4']

         IMG_SIZE = 100
         training_data = []

         def create_training_data():
             for category in CATEGORIES:
                 path = os.path.join(DATADIR,category)
                 class_num = CATEGORIES.index(category) + 1
                 num_of_pics_each = 0
                 for img in os.listdir(path):
                     img_array = cv2.imread(os.path.join(path,img),cv2.IMREAD_GRAYSCALE) #fekete fehér képek beolvasása/színes képek beolv
                     new_array = cv2.resize(img_array,(IMG_SIZE,IMG_SIZE)) #kép átméretezése
                     training_data.append([new_array,class_num])
                     num_of_pics_each+=1
                 print(f"In [[{category}]] found {num_of_pics_each} pics") #kiírjuk hány kép volt minden kategóriában

         create_training_data()
```

Először is fent meg kell adni a DATADIR változóban, hogy a mappák, amik tartalmazzák a különböző képeket hol találhatók, majd a CATEGORIES tömbben fel kell sorolni a mappák nevét jelen esetben ugye a 4 fázis. A beolvasás során ezeket egyesítjük és így olvassuk be a

képeket minden mappából külön-külön. Egy for ciklussal végig megyünk az összes kategórián, egyesítjük az elérési útvonallal és beolvassuk az ott található összes képet. Majd ezt egy tömbben tároljuk, ahol a kép mellett a kategória indexet is megadjuk innen tudjuk, hogy a kép melyik fázisból származik. Itt fekete fehéren olvassuk be a képeket, színes beolvasásnál 3 érték keletkezik ugye az RGB-nek megfelelően, de jelen esetben csak 1. Ez a c2.imread függvényben tudjuk megadni, most cv2.IMREAD_GREYSCALE van megadva ezért fekete fehér. Azért, hogy a különböző méretű képek ne zavarjanak meg és hogy csökkentsük az adat mennyiséget minden képet átméretezek jelen esetben 100x100-asra.

Ezek után „összerázzuk" a tömböt, hogy a különböző kategóriák ne egymás után következzenek a tömbben mert az a tanulás szempontjából káros, mivel ilyenkor azt tanulja meg a modell, hogy mindig 1-es fázis utána azt, hogy mindig 2-es és így tovább.

Jelen esetben szükség van arra, hogy Flatten-eljük a tömböt mert ez a modell csak 1D-s tömböt fogad el.

```
In [4]: target = []
        data = []

        for features, label in training_data:
            data.append(features.flatten())  #a flatten szükséges a model betanításához mivel ez a model csak 1d-s tömböt fogad el
            target.append(label)
```

Itt **RandomForestClassifier** modellt használunk.

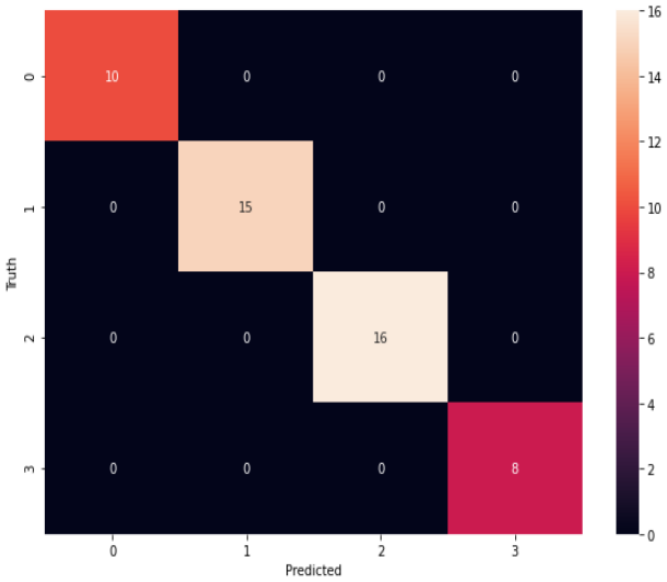
```
In [6]: model.fit(X_train,Y_train)
Out[6]: RandomForestClassifier()

In [7]: model.score(X_test, Y_test) #pontosság
Out[7]: 1.0
```

És látszik, hogy egy ilyen egyszerű modellnél 100% pontosságot érünk el, nagyon jól teljesített. Ezt meg is tudjuk tekinteni a Confusion mátrixon:

In [8]:
```
# nézzük meg az eredményt a confusion mátrixban
Y_predicted = model.predict(X_test)
```

In [9]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test,Y_predicted)
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sn
plt.figure(figsize=(10,7))
sn.heatmap(cm, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Out[9]: Text(69.0, 0.5, 'Truth')



Feladat ismertetése és bemutatása

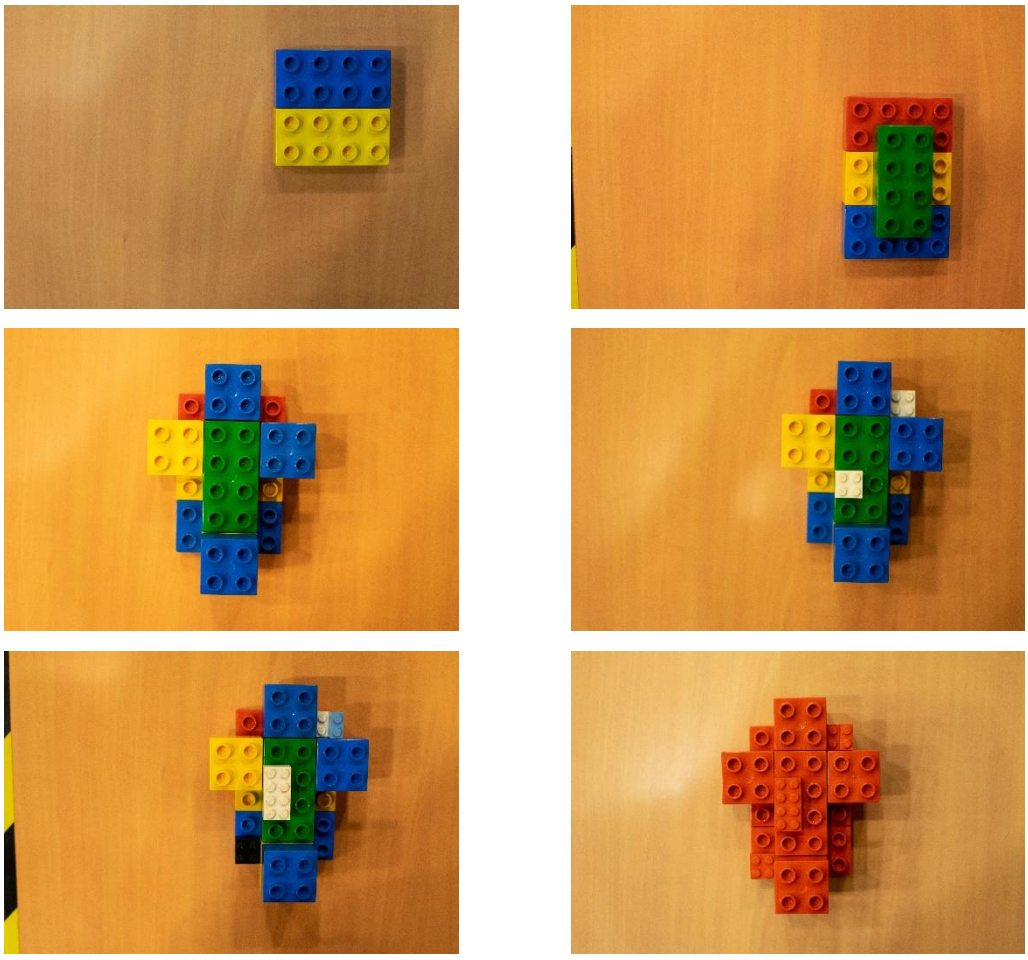
A félév során, építettünk magunknak különböző Lego darabokból pár kisebb építményt, hogy tovább tudjuk fejleszteni az osztályozással kapcsolatos tudásunkat ameddig nincsenek valós képek a munkadarabokra.

Az ötlet az volt, hogy különböző osztályt képzünk az építményekből és erre próbálunk minél jobb modellt építeni. A projekt eleje azzal kezdődött, hogy csináltunk 6 különböző osztályt és ezekre próbáltuk felépíteni a modellünket, majd ahogy haladtunk előre az az ötletünk jött, hogy egy kicsi más oldalról közelítjük meg a dolgot.

A következő megfontolás az volt, hogy csak 3 osztályt képezünk, amelyek között apró az eltérés, egy kicsi sárga airsoft golyó az egyik Lego darabka egyik lyukában. A második fázis képei, 4 különböző háttér előtt, 4 különböző eszközzel készültek, összesen majdnem 1800 kép.

A különböző feladatokat, különböző módszerekkel is megnéztük. Az elején kezdtünk a RandomForest-tel majd ezt követte a Deep Learning-es módszerek.

A feladatunk első fázisa (6 osztály):



RandomForestClassifier

A modell létrehozása:

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
```

A modell pontossága tanítás után:

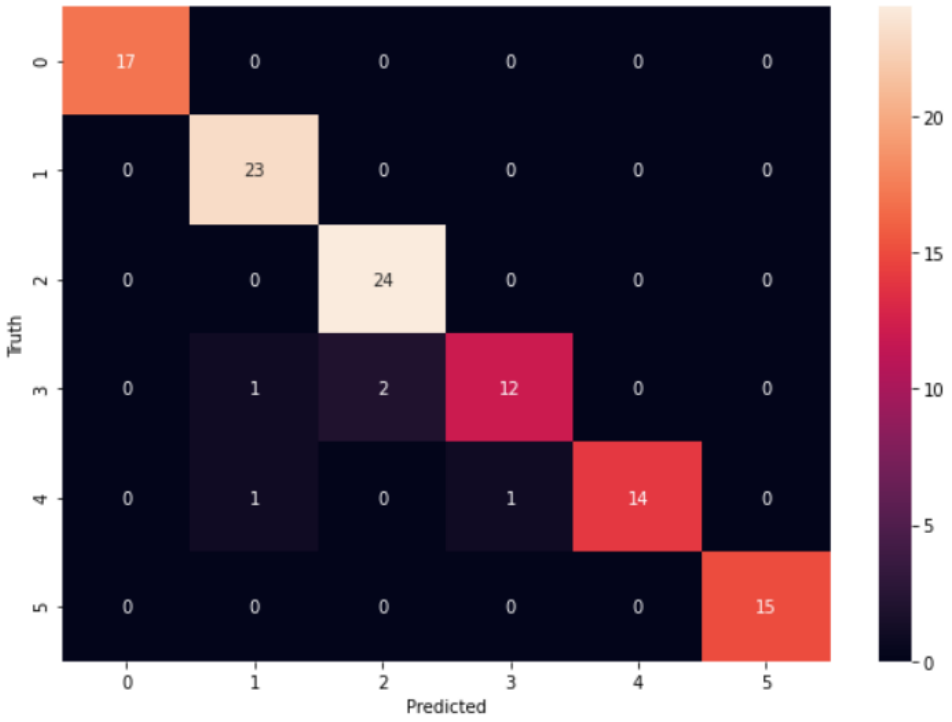
```
In [13]: model.score(X_test, Y_test)
Out[13]: 0.9545454545454546
```

A modell körülbelül 95%-os pontosságot ért el a képeinken, ami nem rossz, ellenben ezek meglehetősen egyszerű képek voltak.

Confusion mátrix:

```
In [17]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn as sn
         plt.figure(figsize=(10,7))
         sn.heatmap(cm, annot=True)
         plt.xlabel('Predicted')
         plt.ylabel('Truth')
```

```
Out[17]: Text(69.0, 0.5, 'Truth')
```



Jól látszik, hogy a modell tényleg a 2. és 3. fázisnál hibázott a legtöbbet, ahol a legkisebb volt a különbség, szám szerint kettő.

Összefoglalva a modell jól teljesített, 95.5%-os sikerességgel tippelt és nagyon egyszerű volt felépíteni, a tanulás is gyorsan ment.

Megjegyzés, hogy a képek nagyon egyszerűek voltak azért, a második fázis végén ugyanez a modell a nehezebb képeken csak 91%-ot ér el és még azok sem tekinthetők a legnehezebbnek, ami azt jelenti érdemes más módszert alkalmaznunk a magas hatékonyság eléréséhez.

Deep Learning

Kipróbált modell:

In [4]:
```
model = tf.keras.models.Sequential([tf.keras.layers.Conv2D(16,(3,3),activation ='relu', input_shape =(200,200,3)),
                                    tf.keras.layers.MaxPool2D(2,2),
                                    #
                                    tf.keras.layers.Conv2D(32,(3,3),activation = 'relu'),
                                    tf.keras.layers.MaxPool2D(2,2),
                                    #
                                    tf.keras.layers.Conv2D(64,(3,3),activation = 'relu'),
                                    tf.keras.layers.MaxPool2D(2,2),
                                    #
                                    tf.keras.layers.Flatten(),
                                    #
                                    tf.keras.layers.Dense(512, activation = 'relu'),
                                    #
                                    tf.keras.layers.Dense(5,activation = 'softmax')
])
```

In [5]:
```
model.compile(loss = 'categorical_crossentropy', optimizer = "adam", metrics = ['accuracy'])
```

In [6]:
```
model_fit = model.fit(train_dataset,
                      steps_per_epoch = 6,
                     epochs=50,
                     validation_data= validation_dataset)
```

A modellünk itt már pár rétegből áll, lássuk ez, hogyan teljesít a feladatunkon. Annyi kis javítás van, hogy a modell majdnem ugyanerre a feladatra készült csak az utolsó osztály nem volt benne ezért az utolsó Dense rétegen a neutronok száma 5 a 6 helyett.

A modell eredménye a következő oldalpn látható.

```
In [6]: model_fit = model.fit(train_dataset,
                              steps_per_epoch = 50,
                              epochs=15,
                              validation_data= validation_dataset)
```

```
Epoch 1/15
50/50 [==============================] - 112s 2s/step - loss: 1.6123 - accuracy: 0.3800 - val_loss: 0.9875 - val_accuracy: 0.61
11
Epoch 2/15
50/50 [==============================] - 102s 2s/step - loss: 0.8568 - accuracy: 0.6842 - val_loss: 0.7522 - val_accuracy: 0.69
75
Epoch 3/15
50/50 [==============================] - 108s 2s/step - loss: 0.7166 - accuracy: 0.7773 - val_loss: 0.6264 - val_accuracy: 0.70
99
Epoch 4/15
50/50 [==============================] - 111s 2s/step - loss: 0.5270 - accuracy: 0.8200 - val_loss: 0.3679 - val_accuracy: 0.88
27
Epoch 5/15
50/50 [==============================] - 107s 2s/step - loss: 0.2417 - accuracy: 0.9433 - val_loss: 0.4148 - val_accuracy: 0.85
19
Epoch 6/15
50/50 [==============================] - 109s 2s/step - loss: 0.2555 - accuracy: 0.9312 - val_loss: 0.3671 - val_accuracy: 0.90
12
Epoch 7/15
50/50 [==============================] - 100s 2s/step - loss: 0.1146 - accuracy: 0.9560 - val_loss: 0.5198 - val_accuracy: 0.82
72
Epoch 8/15
50/50 [==============================] - 98s 2s/step - loss: 0.1845 - accuracy: 0.9480 - val_loss: 0.2480 - val_accuracy: 0.913
6
Epoch 9/15
50/50 [==============================] - 107s 2s/step - loss: 0.1336 - accuracy: 0.9800 - val_loss: 0.2672 - val_accuracy: 0.92
59
Epoch 10/15
50/50 [==============================] - 95s 2s/step - loss: 0.0452 - accuracy: 0.9879 - val_loss: 0.5328 - val_accuracy: 0.895
1
Epoch 11/15
50/50 [==============================] - 107s 2s/step - loss: 0.1150 - accuracy: 0.9717 - val_loss: 0.2706 - val_accuracy: 0.91
98
Epoch 12/15
50/50 [==============================] - 99s 2s/step - loss: 0.0554 - accuracy: 0.9757 - val_loss: 0.1510 - val_accuracy: 0.944
4
Epoch 13/15
50/50 [==============================] - 98s 2s/step - loss: 0.0521 - accuracy: 0.9920 - val_loss: 0.1505 - val_accuracy: 0.944
4
Epoch 14/15
50/50 [==============================] - 102s 2s/step - loss: 0.0474 - accuracy: 0.9838 - val_loss: 0.1720 - val_accuracy: 0.95
68
Epoch 15/15
50/50 [==============================] - 98s 2s/step - loss: 0.0118 - accuracy: 1.0000 - val_loss: 0.1410 - val_accuracy: 0.969
1
```

A modellünk nagyon szép majdnem 97%-os sikerességgel tippelt míg a val_losst is sikerült 0.141-re leredukálni. A modellben minden rétegen az aktivációs függvény a relu kivéve az utolsó rétegen, ahol egy softmax található.

Ez valamennyire bevett dolog majd látszik, hogy a relu az egyik legtöbbet alkalmazott aktivációs réteg továbbá a softmax is sokszor szerepel az utolsó réteg aktivációjaként.

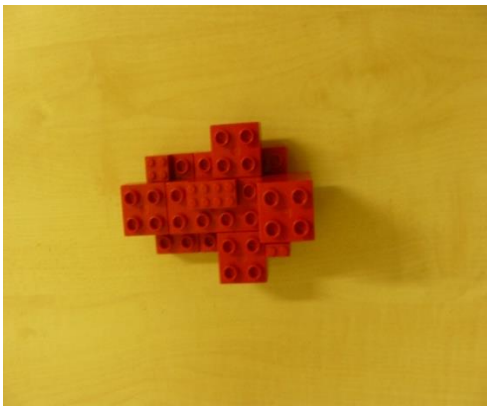
Az optimizer az adam lett, ami megint csak nagyon gyakori, későbbi modelleken látszik, hogy szinte mindenki azt használja, mert gyors és jó eredményeket lehet elérni vele.

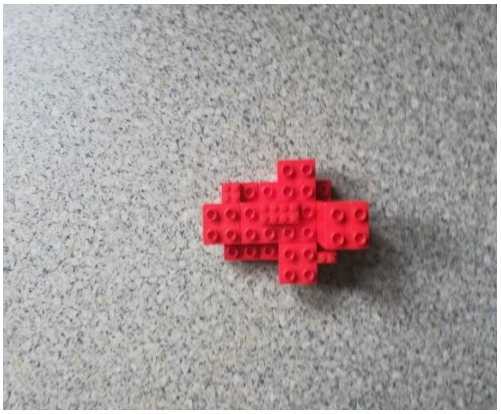
A Flatten réteg csak annyit csinál, hogy 1D-re alakítja a képeinket, amelyek így tudnak majd a Dense rétegre kerülni.

Rengeteg optimalizációs eszközünk van, amelyekből itt alig volt használva, ami azt jelenti, hogy a közel 100%-os pontosság abszolút elérhető kis javításokkal továbbá egy két másik módszer alkalmazásával.

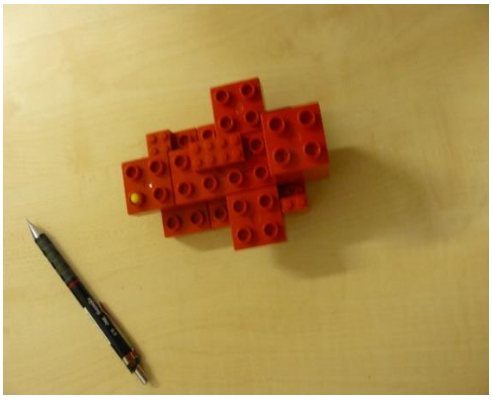
A feladatunk második fázisa (3 osztály):

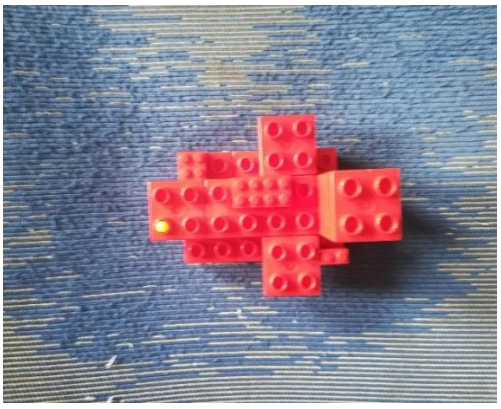
Itt már csak 3 osztályunk van, több képpel viszont nehezebb dolga van a modellünknek.



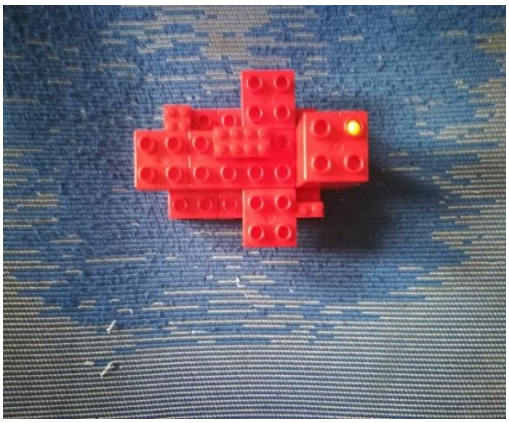


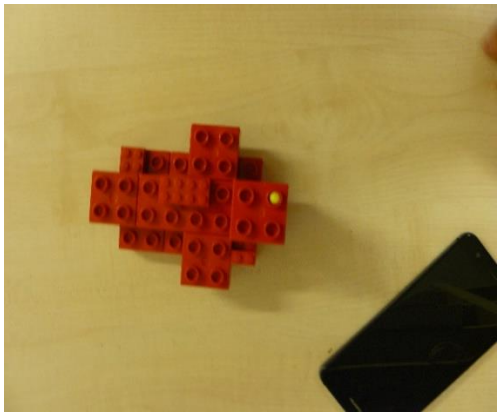
Ezen az osztályon csak egy a piros elemekből összeépített darab található. Próbálkoztunk úgy készíteni a képeket, hogy mindegyiket egy picit más szögből egy picit más háttérrel, elforgatva, egy kicsit beleközelítve, hogy nehezebb legyen a modellünknek, de jobb munkát végezhessünk. Itt is van, ami szürke, háttérnél készült, van amelyik kicsit elmosódott.





Ezen az osztályon annyi lett a változtatás, hogy egy pici sárga bogyó került egy fenti Lego darab egyik lyukába, de az alap piros elemekből álló építmény ugyanaz maradt. Itt is a képek kicsit más szögből más fénynél lettek lefényképezve, más hátérrel és környezettel.





Ennél az osztálynál a változtatás abban rejlik, hogy a sárga bogyó a darab egyik másik részének a lyukába került, de a piros építmény változatlan maradt. Itt is lettek képek, kék háttérnél szürke háttérnél és forgatva, mint a többi osztálynál.

RandomForestClassifier

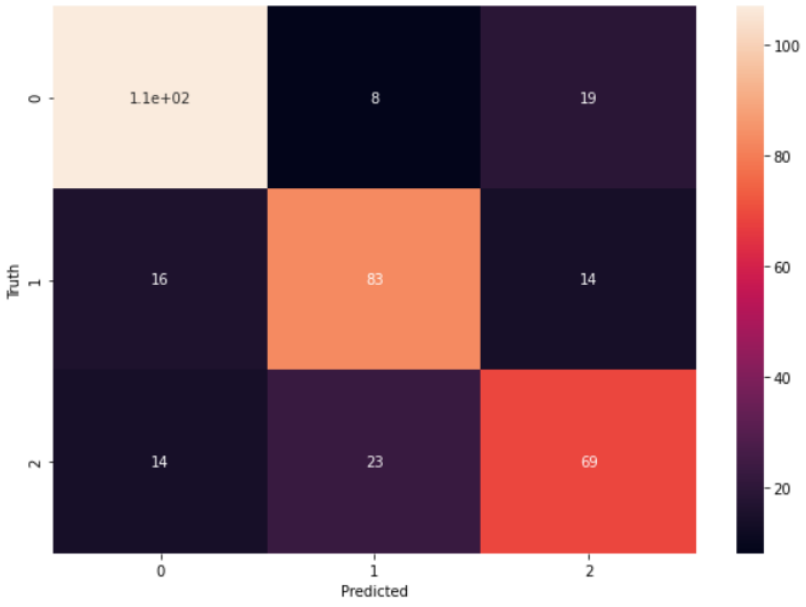
Ennek a nagyon egyszerű modellnek a megalkotása:

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
```

A modell pontossága futtatás után:

```
In [20]: model.score(X_test, Y_test)
Out[20]: 0.9121813031161473
```

Confusion mátrix:



A confusion mátrixon jól látszik, a modell a legtöbb hibát úgy vétette, hogy a 2. és 3. fázisokat keverte össze, ahol az airsoft bogyók más helyen voltak. A második legtöbb hiba az 1. és a 3. fázisok között volt.

Összefoglalva, ez az jelenti, hogy a RandomForestClassifier körülbelül 91%-os pontossággal találja el, hogy az adott képek melyik fázishoz tartozik, ami nem olyan rossz, ahhoz képest, hogy a modell milyen egyszerű. Sajnos ennek a javítása az egyszerűségéből kifolyólag nehéz, esetünkben jobb eredmények után kutatva érdemesebb a problémát más módszerrel megoldani.

Deep learning

A modell felépítése:

```
import tensorflow as tf
from tensorflow.keras.optimizers import RMSprop

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(16,(3,3),activation ='relu', input_shape =(IMG_SIZE,IMG_SIZE,3)))
model.add(tf.keras.layers.MaxPool2D(2,2))
model.add(tf.keras.layers.Conv2D(32,(3,3),activation ='relu'))
model.add(tf.keras.layers.MaxPool2D(2,2))
model.add(tf.keras.layers.Conv2D(64,(3,3),activation ='relu'))
model.add(tf.keras.layers.MaxPool2D(2,2))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(512, activation = tf.nn.relu))
model.add(tf.keras.layers.Dense(4, activation = tf.nn.softmax))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs = 20)
```

Itt is ahogy már az előbbiekben láttuk a szokásos relu aktivációs függvény kerül elő továbbá az utolsó rétegen a softmax.
Ebben a modellben is Dense réteg az utolsó, ami előtt a Flatten() 1D-s adatot állít elő nekünk.
A szokásos adam optimizer van használva itt is továbbá az utolsó réteg egy softmax.
Ennél a modellnél sem volt augmentáció alkalmazva vagy más módszer, ami javítaná a pontosságot viszont így is kiváló eredményeket ért el.

A modell eredménye:

```
Epoch 1/20
50/50 [==============================] - 6s 123ms/step - loss: 26.0865 - accuracy: 0.4183
Epoch 2/20
50/50 [==============================] - 6s 122ms/step - loss: 0.7575 - accuracy: 0.6511
Epoch 3/20
50/50 [==============================] - 6s 119ms/step - loss: 0.5364 - accuracy: 0.7584
Epoch 4/20
50/50 [==============================] - 6s 120ms/step - loss: 0.4386 - accuracy: 0.8189
Epoch 5/20
50/50 [==============================] - 6s 119ms/step - loss: 0.3314 - accuracy: 0.8763
Epoch 6/20
50/50 [==============================] - 6s 119ms/step - loss: 0.2230 - accuracy: 0.9167
Epoch 7/20
50/50 [==============================] - 6s 120ms/step - loss: 0.1956 - accuracy: 0.9268
Epoch 8/20
50/50 [==============================] - 6s 120ms/step - loss: 0.1705 - accuracy: 0.9388
Epoch 9/20
50/50 [==============================] - 6s 121ms/step - loss: 0.1056 - accuracy: 0.9628
Epoch 10/20
50/50 [==============================] - 6s 119ms/step - loss: 0.1057 - accuracy: 0.9609
Epoch 11/20
50/50 [==============================] - 6s 119ms/step - loss: 0.0725 - accuracy: 0.9741
Epoch 12/20
50/50 [==============================] - 6s 122ms/step - loss: 0.0541 - accuracy: 0.9842
Epoch 13/20
50/50 [==============================] - 6s 120ms/step - loss: 0.0224 - accuracy: 0.9937
Epoch 14/20
50/50 [==============================] - 6s 119ms/step - loss: 0.0148 - accuracy: 0.9968
Epoch 15/20
50/50 [==============================] - 6s 120ms/step - loss: 0.0029 - accuracy: 1.0000
Epoch 16/20
50/50 [==============================] - 6s 121ms/step - loss: 0.0013 - accuracy: 1.0000
Epoch 17/20
50/50 [==============================] - 6s 120ms/step - loss: 8.9517e-04 - accuracy: 1.0000
Epoch 18/20
50/50 [==============================] - 6s 119ms/step - loss: 5.0088e-04 - accuracy: 1.0000
Epoch 19/20
50/50 [==============================] - 6s 120ms/step - loss: 3.7098e-04 - accuracy: 1.0000
Epoch 20/20
50/50 [==============================] - 6s 120ms/step - loss: 3.1367e-04 - accuracy: 1.0000
```

A modellünk volt, hogy elérte a 100%-os pontosságot, ami nagyon jónak mondható és ez az eredmény még bőven fejleszthető olyan értelemben, hogy nehezebb képeken is megállja majd a helyét. Itt látszik, hogy ez a modell és az a modell, amit a 6 osztálynál alkalmaztunk meglehetősen hasonló csak itt alkalmaztunk augmentációt továbbá több képünk volt. Rengeteg módszer van ahogy tudunk növelni a pontosságon itt tényleg csak a kezdetek lett bemutatva.

Augmentáció

Mit értünk augmentáció alatt?

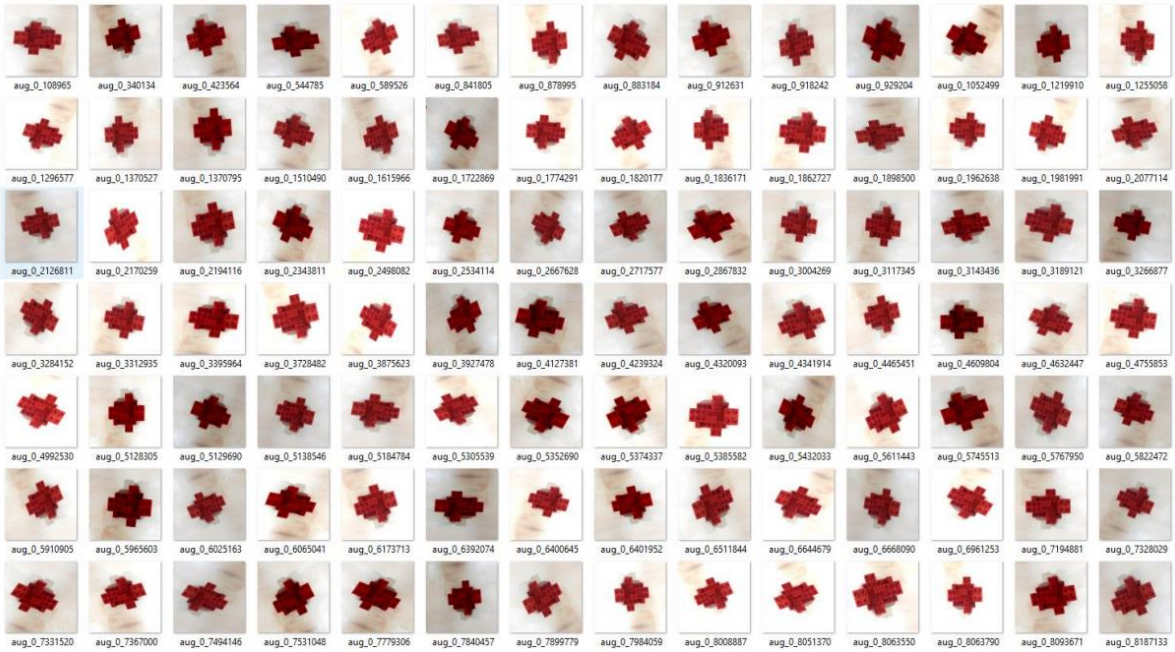
Az adathalmazunkat augmentálhatjuk különböző könyvtárak segítségével. A mi esetünkben az adatok képek voltak, augmentáció segítségével tulajdonképpen „új” képeket generáltunk a régiekből. Az így generált képek paraméterein változtathatunk, erre könyvtáranként eltérő lehetőségeink vannak, de például lehetséges a tükrözés, forgatás, véletlenszerű zoomolás és még folytathatnánk – de ezek pontosabb alkalmazására később még alaposabban kitérünk. Ízelítőnek alább látható egy augmentált képsorozat, ezek alapjául ugyanaz a kép szolgált és a PyTorch RandomResizedCrop tranform-jával augmentálta egyikünk.

RandomResizedCrop működése

Miért augmentálnánk egyáltalán az adathalmazt?

Sokszor előfordul, hogy az adathalmaz mennyisége nem kellően nagy ahhoz, hogy egy általánosítani képes modellt betanítsunk – fennállhat az overfitting jelensége. A mi esetünkben, ahol legjobb esetben is csak nagyjából 600 kép állt rendelkezésünkre, sajnos overfitting nélkül egy adott pontosságot nem tudtunk meghaladni. Ha viszont augmentálunk is, mindjárt más a helyzet.

Augmentáció segítségével előállítható tulajdonképpen tetszőleges mennyiségű kép – bár persze ennek vannak korlátai. Mindenesetre magabiztosan mondhatjuk, hogy egész különböző képek is kijöhetnek eredményül.



Fent látható sorozat ugyanabból a képből készített augmentált képek – sok kép mintha teljesen más lenne, legalábbis a modell a tanítása alatt így „gondolhatja”.

Augmentációhoz többféle API-t is kipróbáltunk, név szerint a PyTorch-ot és a Keras-t (Tensorflow), mindegyiknél bebizonyosodott, hogy ez javít az osztályozás pontosságán. Mivel többen is foglalkoztunk augmentációval a projekt során – és nagyon hasonló eredményekre jutottunk – csak egy részletét mutatnánk be a tényleges futtatások eredményének és inkább a konklúziót tárgyalnánk részletesebben.

Tapasztalatok

A Keras API-n belül az ImageDataGenerator segítségével augmentáltuk a képeket, alább láthatunk erre példát (természetesen vannak további paraméterek is).

```
train_datagen = ImageDataGenerator(rescale = 1./255, rotation_range= 30, width_shift_range = 0.05, height_shift_range  =0.05,
                    zoom_range = [0.8, 1.2], brightness_range = [0.8, 1.2],
                    channel_shift_range= 60, horizontal_flip = True, vertical_flip = True, fill_mode = "reflect")
```

Ezekkel a paraméterekkel generált egyikünk képeket hibakeresésre használt adathalmazból.

Az ImageDataGeneratorral kapcsolatos tapasztalataink

- Nem éri meg nagyon belezoomolni a képbe, olyankor lehet, hogy a teljes alakzat nem is látszódik (a zoom a két tengely mentén nem feltétlenül egyforma, így, ha a közepén is van a lényeges rész, akkor is kilóghat). Ez különösen igaz, ha a shiftelést is használjuk.
- A rescale paramétert minden kép esetén érdemes használni – akár akkor is, ha nem célunk az augmentáció. Ennek oka, hogy a fent látható osztással helyes intervallumra képezhetjük le a színcsatornák értékeit. Ha nem tennénk ezt, rengeteg nagyon durván kiégett képet kapunk eredményül. Lehet, hogy ez valahol nem probléma, mi viszont törekedtünk az augmentált képek „valósághűségére”.
- Az ImageDataGenerator-ral van lehetőségünk lementeni is az augmentált képeket. Ha nem korábban augmentált, majd lementett képekkel tanítunk, mindig új képet generál, amikor beolvas, ezt pedig nem tudjuk utólag kikeresni, sőt, még a tanítást folyamatát is lassítja (persze más esetben korábban generálunk, ami ugyanúgy elég időigényes).

A képeket tanító és validáló részekre bontottuk, one-hot-encode-oltuk.

```
train_datagen = ImageDataGenerator(rescale = 1./255, validation_split = 0.1)


valid_generator = train_datagen.flow_from_directory(directory=path, target_size = (IMG_SIZE,IMG_SIZE),
                      batch_size = 30, classes = ['1','2','3'], subset='validation', color_mode = 'rgb', shuffle = True)
train_generator = train_datagen.flow_from_directory(directory=path, target_size = (IMG_SIZE,IMG_SIZE),
                      batch_size = 30, classes = ['1','2','3'], subset='training', color_mode = 'rgb', shuffle  =True)
```

A fenti képen az immár legenerált képeket olvassuk vissza, így nem kell augmentáció újra. Az eredeti képeknél is a beolvasás hasonló módon történt, de ott 0.2 volt a validation_split értéke.

Az alábbi modellel végeztük a teszteket (néha a kékkel jelölt sorokkal együtt, néha azok nélkül – a tesztek mind az utóbbi módon készültek, hogy ne a modell képességeit, hanem az augmentáció hatásait vehessük gorcső alá).

```
model = Sequential([
    Conv2D(filters = 32, kernel_size = (3,3), activation='relu', padding = 'same', input_shape = (IMG_SIZE,IMG_SIZE,3)),
    MaxPool2D(pool_size=(2,2), strides = 2),
    Conv2D(filters = 64, kernel_size = (3,3), activation='relu', padding = 'same'),
    MaxPool2D(pool_size=(2,2), strides = 2),
    Conv2D(filters = 64, kernel_size = (3,3), activation='relu', padding = 'same'),
    MaxPool2D(pool_size=(2,2), strides = 2),
    Flatten(),
    Dense(units=3, activation='softmax')
])
model.summary()
```

Lássunk pár eredményt

A három fázisból álló hibakeresésre kihegyezett adathalmazra az egyikünk nagyon egyszerű modellje az alábbi eredményt produkálta 20 epoch, vagyis a tanító adathalmazon 20 végigiterálás után:

```
09
Epoch 3/10
53/53 [==============================] - 191s 4s/step - loss: 0.5965 - accuracy: 0.7508 - val_loss: 0.5074 - val_accuracy: 0.78
74
Epoch 4/10
53/53 [==============================] - 179s 3s/step - loss: 0.4353 - accuracy: 0.8202 - val_loss: 0.4082 - val_accuracy: 0.81
61
Epoch 5/10
53/53 [==============================] - 182s 3s/step - loss: 0.3717 - accuracy: 0.8549 - val_loss: 0.3591 - val_accuracy: 0.82
76
Epoch 6/10
53/53 [==============================] - 177s 3s/step - loss: 0.2680 - accuracy: 0.9060 - val_loss: 0.2720 - val_accuracy: 0.88
51
Epoch 7/10
53/53 [==============================] - 177s 3s/step - loss: 0.2316 - accuracy: 0.9066 - val_loss: 0.2951 - val_accuracy: 0.87
93
Epoch 8/10
53/53 [==============================] - 176s 3s/step - loss: 0.2396 - accuracy: 0.8978 - val_loss: 0.2730 - val_accuracy: 0.87
93
Epoch 9/10
53/53 [==============================] - 176s 3s/step - loss: 0.1631 - accuracy: 0.9426 - val_loss: 0.2090 - val_accuracy: 0.90
80
Epoch 10/10
53/53 [==============================] - 176s 3s/step - loss: 0.1518 - accuracy: 0.9432 - val_loss: 0.3127 - val_accuracy: 0.86
78

<tensorflow.python.keras.callbacks.History at 0x136cd44a790>
```

```
model.fit(train_generator, validation_data = valid_generator, epochs = 10)
```

```
Epoch 1/10
53/53 [==============================] - 177s 3s/step - loss: 0.1606 - accuracy: 0.9388 - val_loss: 0.1814 - val_accuracy: 0.89
66
Epoch 2/10
53/53 [==============================] - 177s 3s/step - loss: 0.1564 - accuracy: 0.9338 - val_loss: 0.1180 - val_accuracy: 0.96
55
Epoch 3/10
53/53 [==============================] - 181s 3s/step - loss: 0.1098 - accuracy: 0.9590 - val_loss: 0.1482 - val_accuracy: 0.93
68
Epoch 4/10
53/53 [==============================] - 178s 3s/step - loss: 0.1030 - accuracy: 0.9634 - val_loss: 0.1130 - val_accuracy: 0.94
83
Epoch 5/10
53/53 [==============================] - 177s 3s/step - loss: 0.0844 - accuracy: 0.9716 - val_loss: 0.1109 - val_accuracy: 0.93
10
Epoch 6/10
53/53 [==============================] - 178s 3s/step - loss: 0.0820 - accuracy: 0.9710 - val_loss: 0.1333 - val_accuracy: 0.94
83
Epoch 7/10
53/53 [==============================] - 182s 3s/step - loss: 0.0634 - accuracy: 0.9798 - val_loss: 0.0893 - val_accuracy: 0.94
83
Epoch 8/10
53/53 [==============================] - 178s 3s/step - loss: 0.0694 - accuracy: 0.9773 - val_loss: 0.1493 - val_accuracy: 0.95
40
Epoch 9/10
53/53 [==============================] - 176s 3s/step - loss: 0.0875 - accuracy: 0.9653 - val_loss: 0.1428 - val_accuracy: 0.95
40
Epoch 10/10
53/53 [==============================] - 183s 3s/step - loss: 0.1052 - accuracy: 0.9653 - val_loss: 0.1391 - val_accuracy: 0.96
55
```

Legalsó sorban látható, hogy 96,5% pontosságú lett a végére a modell – nem is rossz, megéri egyáltalán augmentálni ennek fényében? A válasz: Igen! (A teszteléshez validation_split = 0.2-t használtunk)

Eredetileg augmentáltuk a képeket, így összesen a korábbi 1672 kép helyett 52800 képpel tanítottuk a – borzasztó egyszerű felépítésű – modellünket. Ennek eredménye látható a következő oldalon.

```
Epoch 1/10
1587/1587 [==============================] - 993s 626ms/step - loss: 0.8010 - accuracy: 0.5896 - val_loss: 0.5634 - val_accurac
y: 0.7331
Epoch 2/10
1587/1587 [==============================] - 954s 601ms/step - loss: 0.4282 - accuracy: 0.8143 - val_loss: 0.1943 - val_accurac
y: 0.9253
Epoch 3/10
1587/1587 [==============================] - 948s 597ms/step - loss: 0.1272 - accuracy: 0.9559 - val_loss: 0.0910 - val_accurac
y: 0.9692
Epoch 4/10
1587/1587 [==============================] - 941s 593ms/step - loss: 0.0635 - accuracy: 0.9795 - val_loss: 0.0572 - val_accurac
y: 0.9822
Epoch 5/10
1587/1587 [==============================] - 939s 592ms/step - loss: 0.0443 - accuracy: 0.9856 - val_loss: 0.0479 - val_accurac
y: 0.9826
Epoch 6/10
1587/1587 [==============================] - 925s 583ms/step - loss: 0.0314 - accuracy: 0.9900 - val_loss: 0.0334 - val_accurac
y: 0.9896
Epoch 7/10
1587/1587 [==============================] - 912s 574ms/step - loss: 0.0280 - accuracy: 0.9906 - val_loss: 0.0408 - val_accurac
y: 0.9862
Epoch 8/10
1587/1587 [==============================] - 862s 543ms/step - loss: 0.0196 - accuracy: 0.9935 - val_loss: 0.0384 - val_accurac
y: 0.9892
Epoch 9/10
1587/1587 [==============================] - 967s 609ms/step - loss: 0.0203 - accuracy: 0.9935 - val_loss: 0.0317 - val_accurac
y: 0.9894
Epoch 10/10
1587/1587 [==============================] - 959s 604ms/step - loss: 0.0181 - accuracy: 0.9941 - val_loss: 0.0415 - val_accurac
y: 0.9854

<tensorflow.python.keras.callbacks.History at 0x20d01477610>
```

Itt már 4-5 tanítási ciklus után kezdi elérni a végső eredményét, 98,5%-ot, de ennél volt jobb is korábban. Ez 2%-os javulást adott összesen.

A 2%-os javulás nem tűnik soknak, de ha kicsit más kontextusba helyezzük ezt a számoz, talán mégis nagyon szembetűnő a változás: a modell kicsit több, mint harmadannyi alkalommal tévesztett csak, mint korábban! Ráadásul ez még fokozható volna tovább is, ugyanis az eredeti képek esetén még több iterálás után elkezdett fellépni az overfitting jelensége (vagyis az már tovább nem tud javulni, ha egy korábban nem látott képet kell osztályoznia), míg a rengeteg képpel tanított modellünk ezek segítségével még 1%-os többletpontosságot ért el, összesen 99,4%-ot.

Még ehhez képest is lehet tovább javítani, ugyanis a használt modell nagyon egyszerű volt és nem használtuk ki a Dropout nyújtotta lehetőségeket sem, aminek a segítségével még további tanításra lenne lehetőségünk az overfitting fellépése nélkül. A tesztek pedig mindössze 100x100-as képekkel készültek – akár még itt is van lehetőség a pontosításra, bár egyikünk tapasztalatai szerint nincs jelentős pontosságnövekedés nagyobb képméretek esetében – legalábbis erre az adathalmazra.

A fenti eredmények és részletek mind TensorFlow és Keras alkalmazásával készültek, ugyanakkor egy másik library, a PyTorch segítségével is hasonló eredményre jutottunk, kisebb léptékekben.

Konklúzió

Mint azt láthattuk, megéri kihasználni az augmentáció adta lehetőségeket. Ez annál drasztikusabb eredményt produkál, minél kisebb az alap adathalmazunk. Ha összekombináljuk az augmentációt a bonyolultabb osztályozó modellel és pár további optimalizációt végzünk, még jobban megközelíthetjük a tökéletes eredményt – de ez persze nagyban függ a képektől is.