Sandu Anastasia

Grupa 242

Brain Anomaly Detection

Scopul acestui proiect este clasificarea unor tomografii in 2 clase :

una care conține anomalii (notată cu 1) și una care este normală (notată cu 0).

Seturile de date pe care le primim sunt reprezentate de:

- data/data– 22.149 de poze, dintre care:

-15.000 de poze de train ale căror label-uri se găsesc in train\_labels.txt

-2.000 de poze de validare ale căror label-uri se gasesc in validation\_labels.txt

-5.149 de poze de test

Modele incercate:

-Naive Bayes

-Random Forest

- Convolutional neural network

NAIVE BAYES

Pentru o primă abordare a problemei am incercat Naïve Bayes. Acesta este un algoritm care se bazează pe Teorema lui Bayes pentru a clasifica datele. Teorema lui Bayes spune că probabilitatea condiționată a unui eveniment A, dat fiind un alt eveniment B, poate fi calculată folosind probabilitatea condiționată inversă a evenimentului B, dat fiind evenimentul A, împreună cu probabilitatea marginală a evenimentelor A și B.

In cazul nostru, clasificarea imaginilor, Naïve Bayes acționează în felul următor: Mai întai se antrenează pe datele de train, extrăgand caracteristicile din imagini și calculează probabilitățile a priori (probabilitatea ca o imagine sa aparțină uneia dintre cele 2 clase (cu anomalie sau fără) înainte de a fi analizată) avand în vedere caracteristicile extrase anterior. Apoi calculează probabilitațile conditionate pentru ambele clase și fiecare caracteristică. În final calculează probabilitațile posterioare ale datelor de test și clasifică fiecare imagine în clasa cu cea mai mare probabilitate.

De precizat este și că Naïve Bayes presupune că fiecare caracteristică sau atribut al imaginii este independent de celelalte și au un impact egal asupra clasificării.

Pentru a citi datele am implementat funcția read care citește o imagine, folosind biblioteca PIL, dintr-un fișier specificat ca parametru, pe care o convertește apoi intr-un obiect numpy array și îl returneaza.

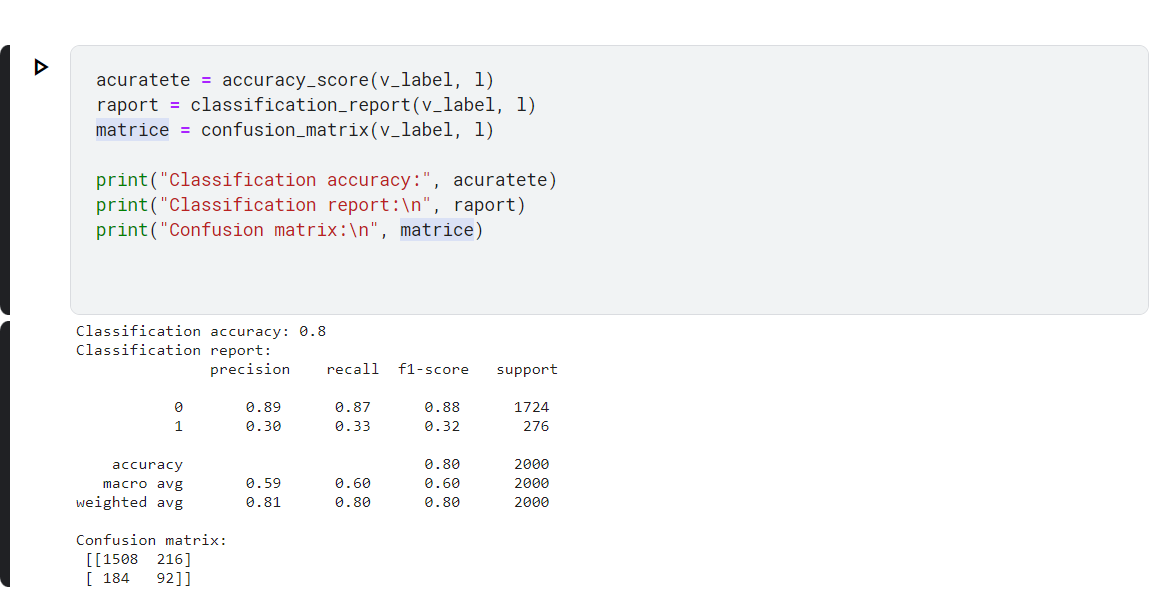
Apoi citesc datele de train, de validare si de test. train\_img este o listă vidă la inceput, la fel si labels. Din fișierul train\_labels.txt citesc pe rand id-ul și eticheta fiecarei poze. Apoi caut imaginea in data/data după id și o citesc cu ajutorul funcției read definită anterior și o adaug în lista train\_img și eticheta în labels. Folosesc variabila cnt pentru a citi doar primele 15.000 de date (cele de training). În final convertesc train\_img într-un numpy array și îl fac 2-dimensional și labels în numpy array și îl fac de date intregi.

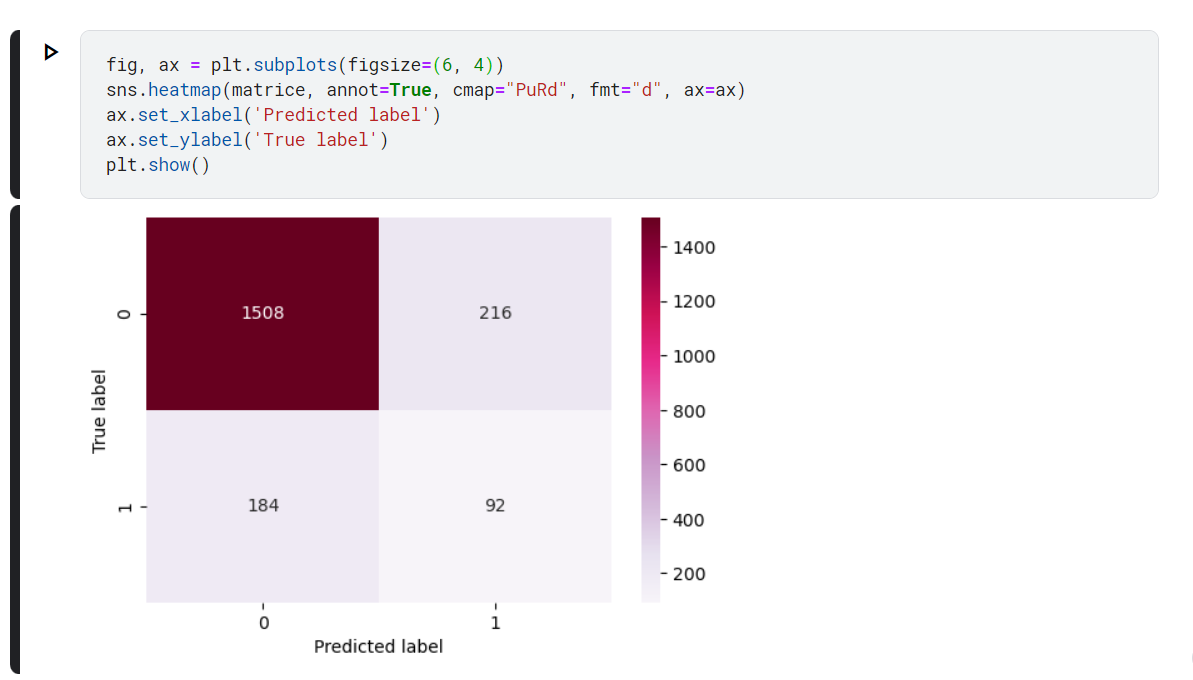
Analog și pentru datele de validare (v\_img si v\_label) și pentru datele de test (fara etichete, v\_imgfinal).

După ce am citit toate datele îmi construiesc modelul folosind GaussianNB din biblioteca scikit-learn și îl antrenez pe datele de train și apoi fac predicția pe validare.

Aceasta este acurațea pe datele de validare:

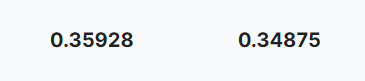






În final am făcut predicția pe datele de test și le-am scris într-un fișier în formatul cerut.

Pe Kaggle am obținut inițial scorul de 0.34875 pe 20% din date iar cel final de 0.35928.



RANDOM FOREST   
Pentru a doua abordare am ales modelul Random Forest care combină mai mulți arbori de decizie pentru o predicție cat mai precisă. Fiecare arbore este antrenat pe o submulțime aleatoare de imagini din care extrage niște caracteristici. Fiecare arbore produce o predicție pentru imaginea de intrare și apoi predicțiile sunt combinate prin votare majoritară pentru clasificare sau prin mediere pentru regresie.

Și în acest model am folosit aceeași citire și aceleași structuri de date de la Naïve Bayes. Am implementat funcția read de care m-am folosit pentru a citi apoi datele de train (train\_img si labels), datele de validare (v\_img si v\_labels) și datele de test (v\_imgfinal).

Apoi construiesc modelul Random Forest folosind RandomForestClassifier din biblioteca scikit-learn.

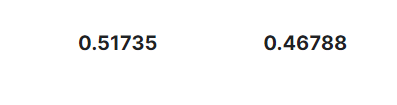
Prima dată am folosit următorii hiperparametrii:

n\_estimators=100, care reprezintă numărul de arbori din padure

max\_depth=10, care reprezintă adancimea maximă a fiecărui arbore

random\_state=42, care controlează aleatorizarea din cadrul modelului și asigură reproducibilitatea rezultatelor

Pe Kaggle am obținut inițial scorul de 0.46788 pe 20% din date iar cel final de 0.51735.



A doua oară am incercat cu valorile:

n\_estimators=130,

max\_depth=20,

random\_state=45

Pe Kaggle obținut inițial initial scorul de 0.44651 pe 20% din date iar cel final de 0.51749.



Apoi am încercat cu următorii hiperparametri:

n\_estimators=130,

max\_depth=20,

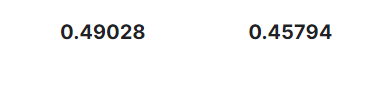
min\_samples\_split=2, care reprezintă numărul minim de exemple necesare pentru a face o nouă împărțire într-un nod

min\_samples\_leaf=1, care reprezintă numărul minim de exemple necesare pentru a forma o frunză într-un nod

max\_features=16, care reprezintă numărul maxim de caracteristici care trebuie luate în considerare la fiecare împărțire

random\_state=45

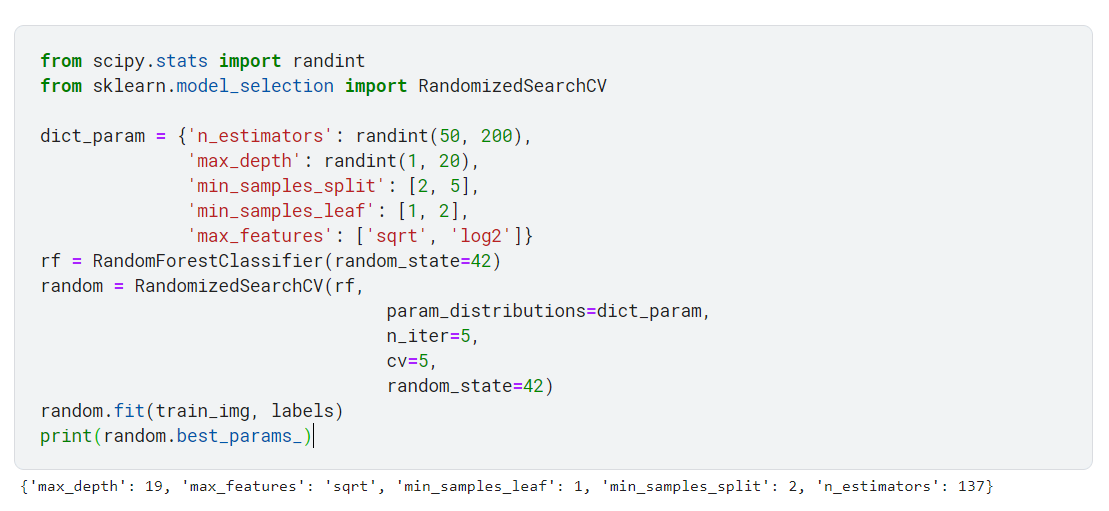
Pe Kaggle am obținut inițial scorul de 0.45794 pe 20% din date iar cel final de 0.49028.



În final am folosit o tehnică de hyperparameter tuning numită Randomized Search Cross Validation pentru a gasi cei mai buni hiperparametri pentru model.

Am definit un dictionar dict\_param care conține intervalul pentru fiecare hiperparametru pentru căutarea aleatorie. Apoi, un obiect rf RandomForestClassifier cu un random\_state fixat la 42.

Am creat un obiect random RandomizedSearchCV care va folosi rf și dict\_param pentru a găsi cei mai buni hiperparametri. Argumentul n\_iter specifică câte combinări aleatorii de hiperparametri să fie încercate, iar cv specifică numărul de fold-uri pentru validarea încrucișată. Random va rula căutarea aleatorie pe datele de train. În final, random.best\_params\_ returneaza combinația de hiperparametri care au dus la cea mai bună performanță a modelului în timpul căutării aleatorii pe care ii afisez.



Dupa ce am folosit hyperparametrii gasiți anterior:

n\_estimators=137,

max\_depth=19,

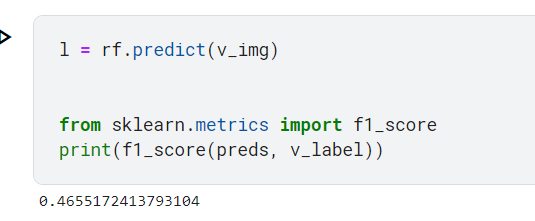
random\_state=42,

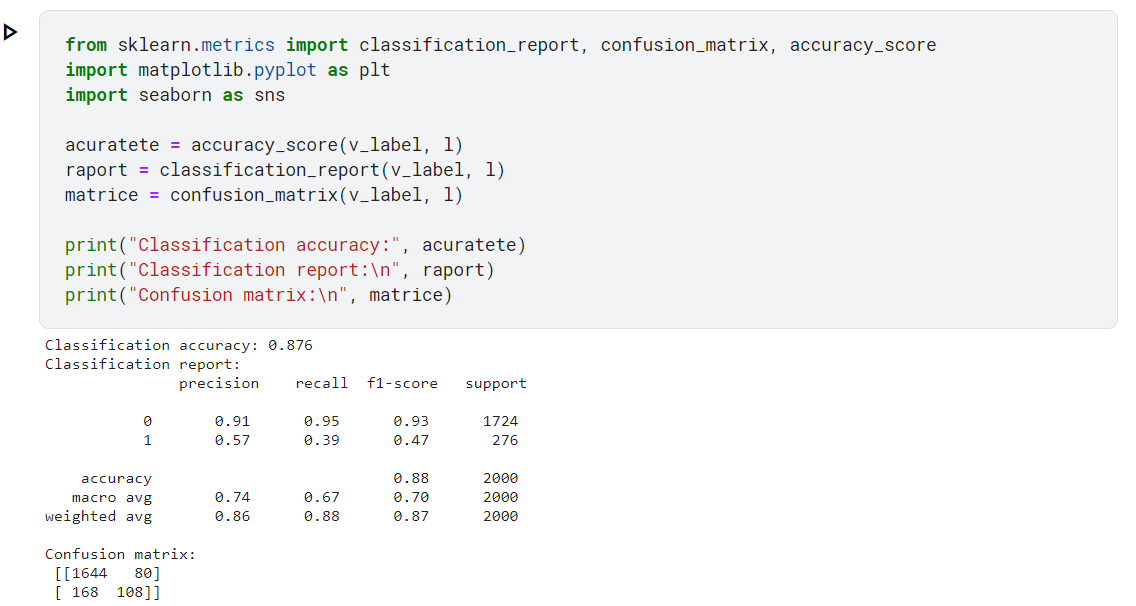
min\_samples\_split=2,

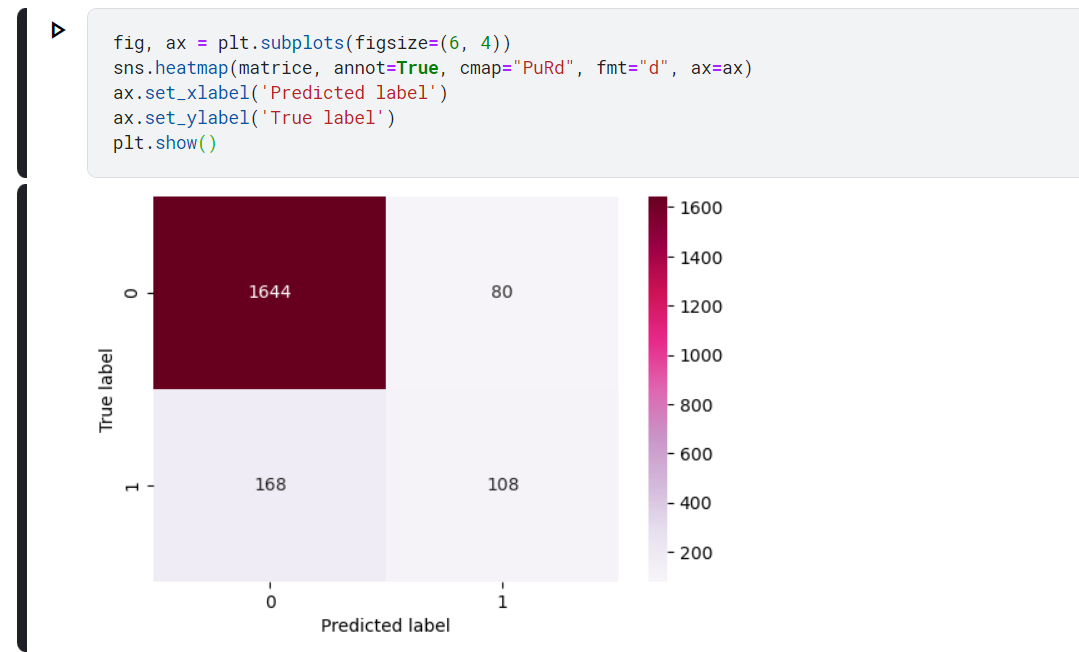
min\_samples\_leaf=1,

max\_features='sqrt'

Am făcut predicția pe datele de validare și acesta este f1 scorul:







Pe Kaggle am obținut inițial scorul de 0.48401 pe 20% din date iar cel final de 0.50786.



CNN

Pentru ultima abordare am încercat modelul CNN, care s-a dovedit a fi cel mai eficient datorită structurii sale speciale construită din mai multe straturi. Acestea folosesc operațiuni de convoluție pentru a detecta caracteristici din imagini, si ulterior pentru a realiza clasificarea.

Am preluat aceeași citire de pană acum și pentru acest model dar cu mici modificari:

-în funcția read de data aceasta redimensionez imaginile (am încercat mai multe dimensiuni, am avut și probleme cu memoria și din această cauza a trebuit să mă rezum la maxim 150 care s-a dovedit a fi și cea mai optimă, dar am încercat și 64, 80,90,100,110,120,128,140)

-după ce am citit datele de train (train\_img și labels), datele de validare (v\_img și v\_labels) și datele de test (v\_imgfinal) toate în nparrays am convertit valorile pixelilor la float64. Am abordat această pre-procesare a datelor pentru a se încadra în intervalul [0,1] deoarece în rețeaua mea am folosit funcțiile Sigmoid și Relu care sunt funcții non-liniare și sunt sensibile la variațiile mari ale valorilor de intrare. Ulterior am încercat și cu float32 pentru a reduce memoria și a îmbunătăți viteza de procesare a datelor.

-de asemenea, in unele variante de cod am normalizat datele folosind funcția preprocess\_input() din biblioteca Keras

-în alte variante am incercat normalizarea datelor prin împărțirea fiecărei valori a pixelilor cu 255

Definirea modelului:

-Conv2D – acest strat aplică filtre asupra inputului care se deplasează prin imagine și calculează produsul scalar între fiecare element din filtru și secțiunea corespunzătoare a imaginii pentru a genera o matrice de feature-uri, care este ulterior prelucrata de alte straturi

-BatchNormalization- acesta ajustează și normalizează activările de la stratul anterior, ajutand la îmbunătățirea stabilității și vitezei de antrenare a rețelei, și la reducerea riscului de overfitting.

- MaxPooling2D- acesta reduce dimensiunile matricii de feature-uri prin selectarea valorilor maxime din regiunile de dimensiune specificată ca parametru

-Flatten- aplatizează matricea de feature-uri rezultată în urma convoluției și reducere dimensiunea într-un vector unidimensional

- Dense-acest strat primeste vectorul aplatizat și îl transformă într-un vector de dimensiunea specificată

- Dropout- acesta reduce overfitting-ul prin eliminarea unor neuroni

Hyperparametrii:

Pentru straturile convoluționale:

-am testat mai multe valori pentru numărul de filtre din mulțimea {8,16,32,64,128}, încercand să plec de la valoarea 8 sau 16 și să cresc din strat in strat numărul de filtre, sau invers, să plec de la o valoare mai mare și să scad treptat numărul filtrelor

-am încercat mai multe variante de kernel (3,3), (5,5), (2,2) dintre care am observat că pentru codul meu cea mai eficientă a fost (3,3) – adică dimensiunea matricii care se plimbă pe imagine și îi aplică filtrele

-am folosit funcția de activare Relu - este o funcție de activare non-liniară care returnează valoarea 0 dacă intrarea este negativă și intrarea însăși dacă este pozitivă. Am ales această funcție deoarece este usoară și eficientă

Pentru pooling:

-am folosit o fereastră de (2,2) – adică dimensiunea unei matrici care se deplasează de-a lungul imaginii de intrare, extrăgând valoarea maximă din acea zonă și creând o imagine de ieșire cu o dimensiune mai mică

Pentru Dropout:

-am încercat mai multe valori din mulțimea {0.3 , 0.4 , 0.5 , 0.6 , 0.7} dintre care am observat că 0,5 a fost cea mai eficientă pentru codul meu deoarece cu 0,3 se facea overfitting si cu 0,7 underfitting

Pentru Dense:

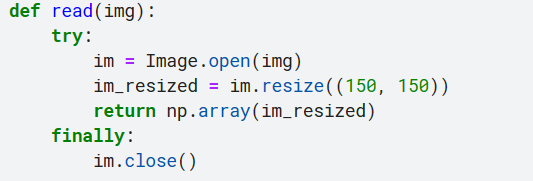
-am încercat cu 16 si 128 de unități dintre care pentru codul meu cel mai eficient a fost 128 și funcția de activare Relu

-pentru ultimul strat Dense am folosit funcția Sigmoid care este o funcție non-liniară ce transformă orice valoare reală într-o valoare între 0 și 1 și o unitate pentru a clasifica in 2 clase

Preprocesarea datelor:

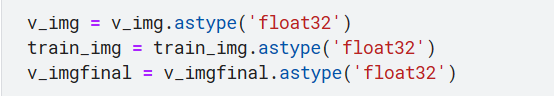
Așa cum am spus și mai sus, am incercat diverse metode de preprocesare a datelor cum ar fi:

- Metoda resize() din biblioteca PIL (Python Imaging Library) care redimensionează o imagine în funcție de dimensiunile specifice ca parametri

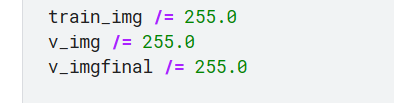


Dimensiunea 150 este dimensiunea finală la care am rămas dar am incercat și cu alte valori precum 64, 80,90,100,110,120,128 si 140.

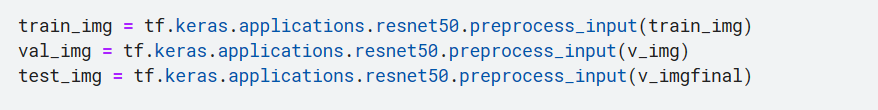
-Metoda astype() din NumPy care convertește tipul de date al unui tablou NumPy în alt tip de date specificat



-Am normalizat datele prin imparțirea valorilor pixelilor la 255, pentru a obține valori intre 0 si 1



-Am folosit funcția preprocess\_input din biblioteca Keras, care efectuează o serie de operații, cum ar fi normalizarea pixelilor imaginii și re-centrarea valorilor acestora



Augmentarea datelor

Dacă în primele variante ale codului nu am folosit deloc augmentarea datelor, am inceput apoi să folosesc pentru a ajuta modelul să invețe caracteristici cat mai generale și a reduce overfitting-ul.

În ultima variantă a codului (cea mai eficientă) am definit două generatoare de imagini pentru datele de train si validare folosind ImageDataGenerator din TensorFlow și am incercat urmatoarele transformări:

-rotation\_range- care rotește aleatoriu imaginea cu un unghi cuprins intre parametri specificați

-width\_shift\_range-care mută aleatoriu imaginea orizontal

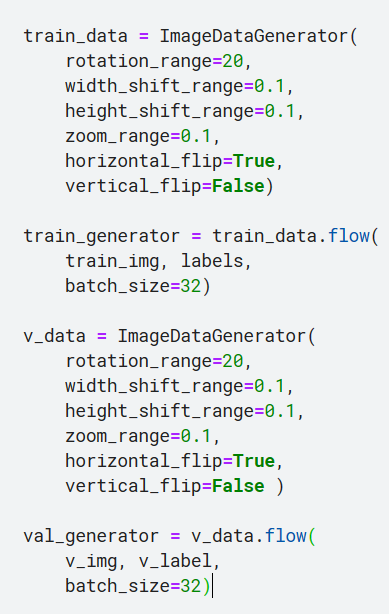
-height\_shift\_range- care mută aleatoriu imaginea vertical

-zoom\_range-care marește sau micșorează aleatoriu imaginea

-horizontal\_flip-care intoarce aleatoriu imaginea pe orizontală

-vertical\_flip-care intoarce aleatoriu imaginea pe verticală

Apoi le folosesc pentru a genera batch-uri de imagini de antrenament și de validare în timpul antrenării modelului



Pentru mărimea batch-urilor am încercat 64 și 32 dar pentru codul meu cel mai eficient a fost 32.

Configurarea modelului:

In toate variantele am folosit:

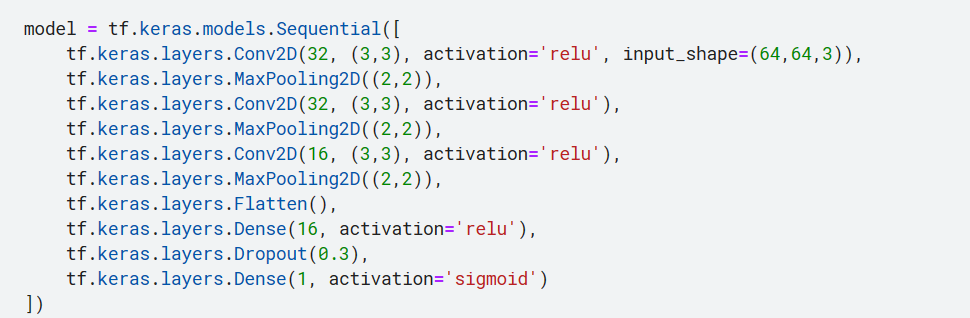
- Optimizer-ul Adam, un algoritm de optimizare a parametrilor, deoarece este eficient și ajută la obținerea unei viteze mai rapide de convergență

-Binary Crossentropy care este o funcție de pierdere folosită în cazul problemelor de clasificare binară

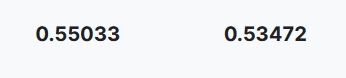
-Accuracy care este o metrică utilizată pentru a evalua performanța modelului de clasificare

Variante incercate:

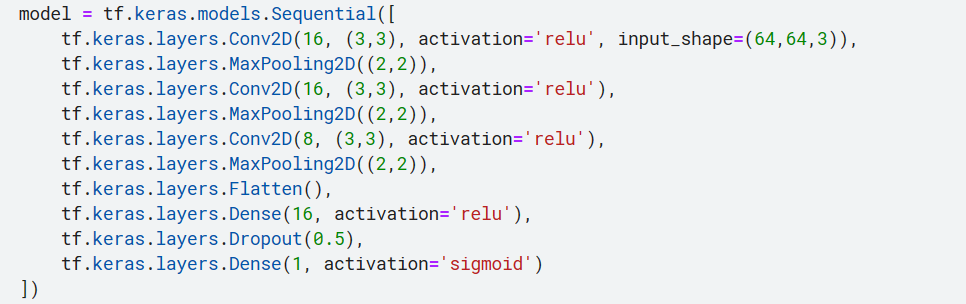
In prima variantă am lucrat cu imaginile redimensionate la 64, am convertit valorile pixelilor la float64 si am folosit functia preprocess\_input și am antrenat modelul pe 10 epoci. Am mers pe o rețea mai mica:



Pe Kaggle am obținut inițial scorul de 0.53472 pe 20% din date iar cel final de 0.55033.



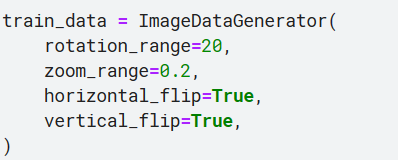
În a doua variantă am lucrat tot cu imaginile de dimensiunea 64\*64, am folosit conversia la float64 și preprocess\_input și am antrenat modelul pe 20 de epoci. Am schimbat rețeaua :



Pe Kaggle am obținut inițial scorul de 0.5418 pe 20% din date iar cel final de 0.539.



În a treia variantă am folosit imaginile tot la 64\*64, conversia la float54 si preprocess\_input și am antrenat modelul pe 20 de epoci. Am încercat și să augmentez datele folosind mai puține transformări:



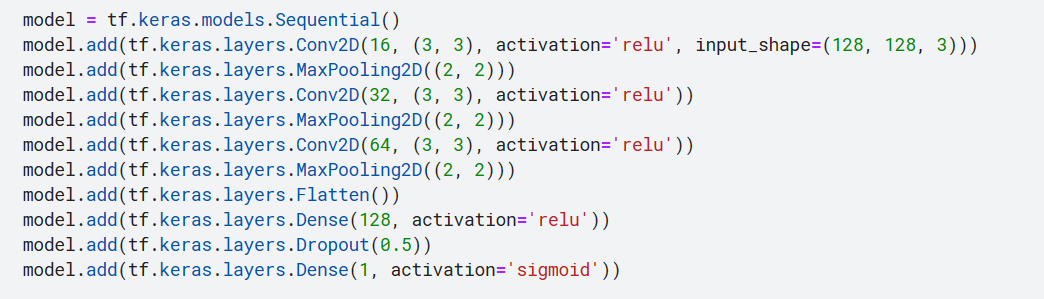
Am folosit următoarea rețea:



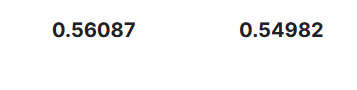
Pe Kaggle am obținut inițial scorul de 0.46975 pe 20% din date iar cel final de 0.49915.



În a patra variantă am lucrat cu imaginile la dimensiunea 128\*128, am folosit conversia la float32, normalizarea prin împărțirea la 255, am antrenat modelul pe 30 de epoci și am folosit urmatoarea rețea:

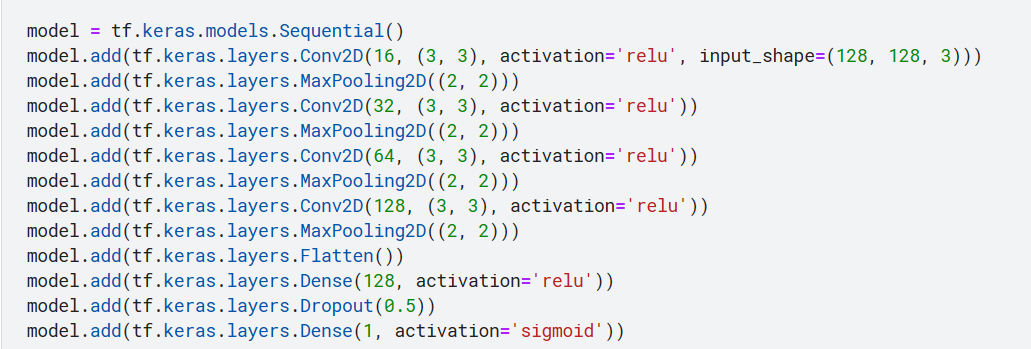


Pe Kaggle am obținut inițial scorul de 0.54982 pe 20% din date iar cel final de 0.56087.

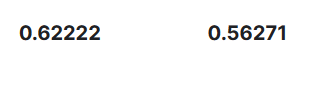


Acestea sunt primele variante pe care le-am incercat. Am avut multe incercări pană să ajung la variantele finale ale codului.

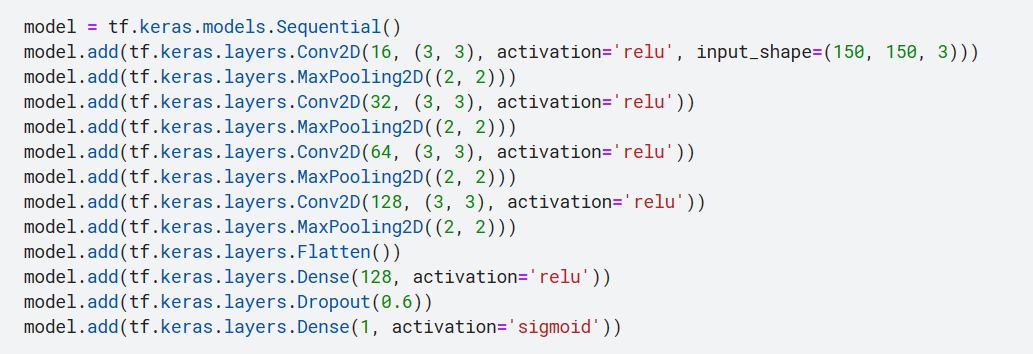
Într-una dintre variantele finale am lucrat cu imaginile la dimensiunile 128\*128, am folosit conversia la float32, normalizarea prin împărțirea la 255, augmentarea datelor cu batch\_size=64 și am antrenat modelul pe 20 de epoci. Aceasta este rețeaua folosită:



Pe Kaggle am obținut inițial scorul de 0.56271 pe 20% din date iar cel final de 0.62222.



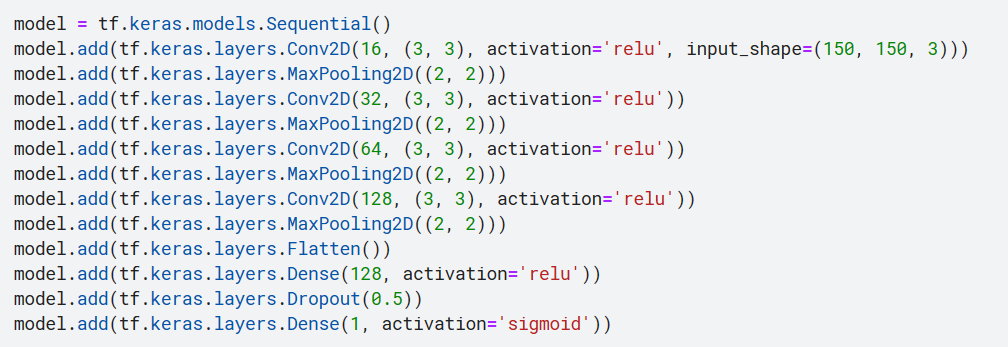
Într-una dintre variantele finale am lucrat cu imaginile de 150\*150, am folosit conversia la tipul float32, normalizarea prin împărțirea la 255, augmentarea datelor cu batch-size=64 și am antrenat modelul pe 30 de epoci. Aceasta este rețeaua folosită:



Pe Kaggle am obținut inițial scorul de 0.58006 pe 20% din date iar cel final de 0.56764.



În varianta finală am lucrat cu imaginile la dimensiunile 128\*128, am folosit conversia la tipul float32, normalizarea prin împărțirea la 255, augmentarea datelor cu batch-size ul de 32 și am antrenat modelul pe 20 de epoci. Aceasta este rețeaua folosită:



Pe Kaggle am obținut inițial scorul de 0.57236 pe 20% din date iar cel final de 0.59006.



Acesta s-a dovedit a fi cel mai eficient cod.

