Brain Anomaly Detection

Olăeriu Vlad Mihai, grupa 241

Aceasta este documentația pentru soluția mea din cadrul competiției "Brain Anomaly Detection", în care a trebuit să construiesc un clasificator, pe un set de date ce conține imagini cu tomografii ale creierului, care decide dacă acel creier are sau nu o anomalie.

Cuprins

1. Inițializarea datelor

Pentru a citi imaginile și etichetele din fișier, m-am folosit de modulul **glob**, cu care am extras toate path-urile corespunzătoare imaginilor și le-am împărțit corespunzător în:

- Imagini de **antrenare**: 15000
- Imagini de **validare**: 2000
- Imagini de **testare**: 5149

Aceste imagini le-am stocat în vectori 4 dimensionali, mărimea fiecărei dimensiuni urmând să fie explicată în secțiunea următoare.

2. Preprocesarea imaginilor

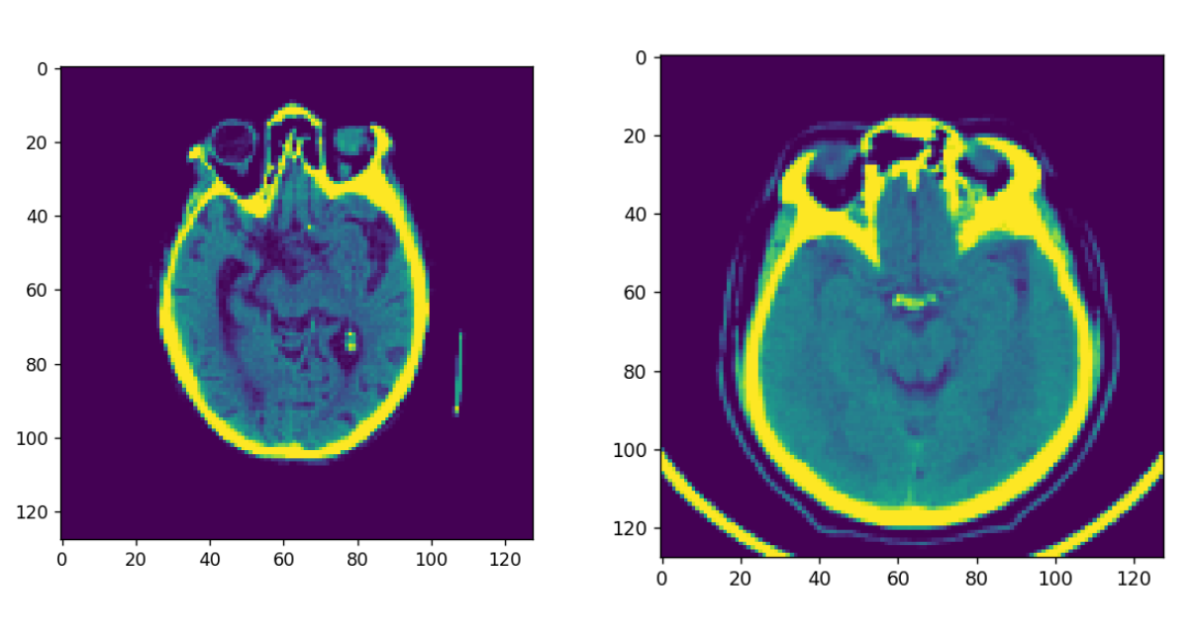
2.1. Sporirea computațională

Pentru a spori viteza computației, am decis să *micșorez imaginile*, transformându-le din formatul 224 x 244 în 128 x 128; de asemenea, deoarece nu culoarea este importantă în clasificarea tomografiilor, ci detectarea de margini și extragerea de caracteristici, am *schimbat canalul de culori* din RGB în grayscale.

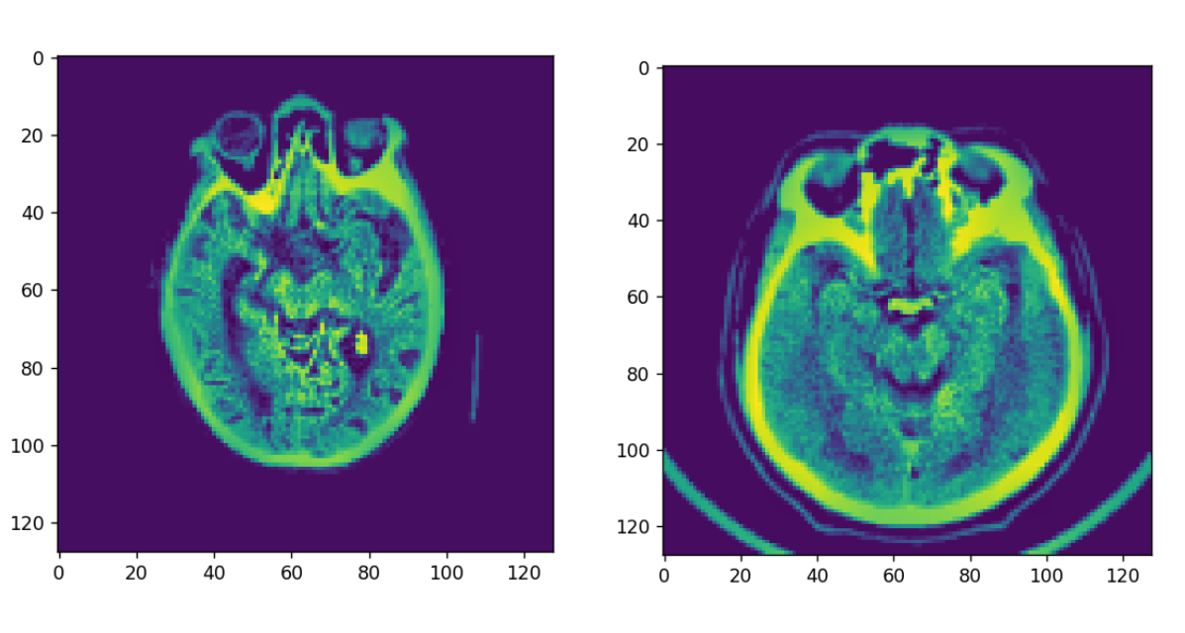
Astfel, vectorii in care am stocat imaginile vor avea forma _(numar_de*imagini, 128, 128, 1)*, unde 128 reprezintă inaltimea si lățimea imaginii, iar 1 numărul canalurilor de culori.

2.2 Ameliorarea zgomotului

În urma modificărilor specificate antrerior, acestea sunt câteva exemple din tomografiile transformate:



După vizualizarea imaginilor, m-am gândit că există prea mult zgomot între pixelii acestora și că acesta trebuie redus. Întâi am aplicat **GaussianBlur-ul**(reduce diferențele mari dintre pixeli), după aceea am decis să măresc și contrastul, deoarece apar anumite umbre pe imagini ce ar trebui îndepărtate. Pentru mărirea contrastului am folosit **CLAHE**(*Contrast Limited Adaptive Histogram Equalization*). După aplicarea filtrelor, imaginile s-au modificat astfel:



Ulterior, după ce m-am documentat mai bine despre tomografiile pe creier, am aflat că pixelii galbeni din imagini reprezintă craniul uman. Acest fapt mi-a demonstrat de ce nu se îmbunătățea acuratețea, dimpotrivă, chiar scădea, în momentul în care foloseam filtrele. Ar trebui să se poată delimita craniul de creier. Din acest motiv am renunțat în a aplica aceste filtre.

2.3 Normalizarea imaginilor

În stagiul preprocesării, am aplicat tipul de normalizare min-max, astfel încât valoarea pixelilor să se încadreze în intervalul [0, 1]. Rolul acestei normalizări este de a avea valorile într-un interval uniform, care ușurează procesarea informației de către model.

Astfel, funcția de citire a imaginilor este:

```
def process_image_CNN(img):
    image = cv2.imread(img)
    image = cv2.resize(image, (128, 128), interpolation=cv2.INTER_AREA)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # image = cv2.GaussianBlur(image, (1,1), 0)
    # image = (cv2.createCLAHE(clipLimit=10)).apply(image)

    return image / 255
```

2.4 Augmentarea imaginilor

Pe parcursul dezvoltării modelelor, când am ajuns la construcția rețelelor neuronale convolutionale, am decis ca înainte să modific arhitectura, ar trebui să încerc să măresc numărul de imagini de antrenare.

Pentru generarea de imagini augmentate, am folosit tool-ul **ImageDataGenerator** de la Keras, care generează un număr fix de imagini etichetate, grupate în batch-uri. Asupra imaginilor standard, sunt aplicate în mod **aleator** diferitele filtre oferite drept parametrii în momentul definirii generatorului.

```
train_datagen = ImageDataGenerator(
        width_shift_range=0.3,
        height_shift_range=0.3,
        horizontal_flip=True,
        shear_range=0.3)
```

Filtrele vor mută imaginile spre stânga/dreapta cu o sensibilitatea maximă de 0.3, le vor întoarce orizontal, sau vor 'forfeca' cu un unghi între [-0.3, 0.3]

3. Dezvoltarea modelelor

3.1 Naive Bayes

3.1.1 Definirea modelului

Primul model pe care l-am construit este cu ajutorul clasificatorului **Naive Bayes**, ce are la baza *teorema probabilității condiționate* a lui Bayes. Acesta este un clasificator simplu, care analizează un vector de caracteristici pentru fiecare imagine, considerând aceste caracteristici independente unele față de celelalte.

```
import numpy as np
from sklearn.naive_bayes import MultinomialNB

def generate_intervals(num_intervals):
    return np.linspace(start=0, stop=256, num=num_intervals)

def transform_images_values_to_bins(images, bins):
    for cnt, image in enumerate(images):
        images[image] = np.digitize(image, bins)
    return images - 1

naive_bayes_model = MultinomialNB()
new_bins = generate_intervals(4)

# training_images ==> vector cu imaginile ce trebuie testate
# training_labels ==> vector cu etichetele fiecarei imagini
# analog validation_images, training_images

x_train = transform_images_values_to_bins(training_images, new_bins)
x_validation = transform_images_values_to_bins(validation_images, new_bins)

naive_bayes_model.fit(x_train, training_labels)
validation_predictions = naive_bayes_model.predict(x_validation)
```

Pentru Naive Bayes nu am normalizat imaginile, deoarece am ales să transform pixelii din valori continue în valori discrete. Având un număr de intervale dorit, fiecare pixel îl asociem intervalului, astfel încât aceștia se vor transformă din **[0, 255]** în **[0, num_intervals]**.

Atunci când numărul de intervale dorit este egal cu 4, vom obține:

	128 x 128
Accuracy	0.591
Precision	0.148
Recall	0.416
F1 score	0.219

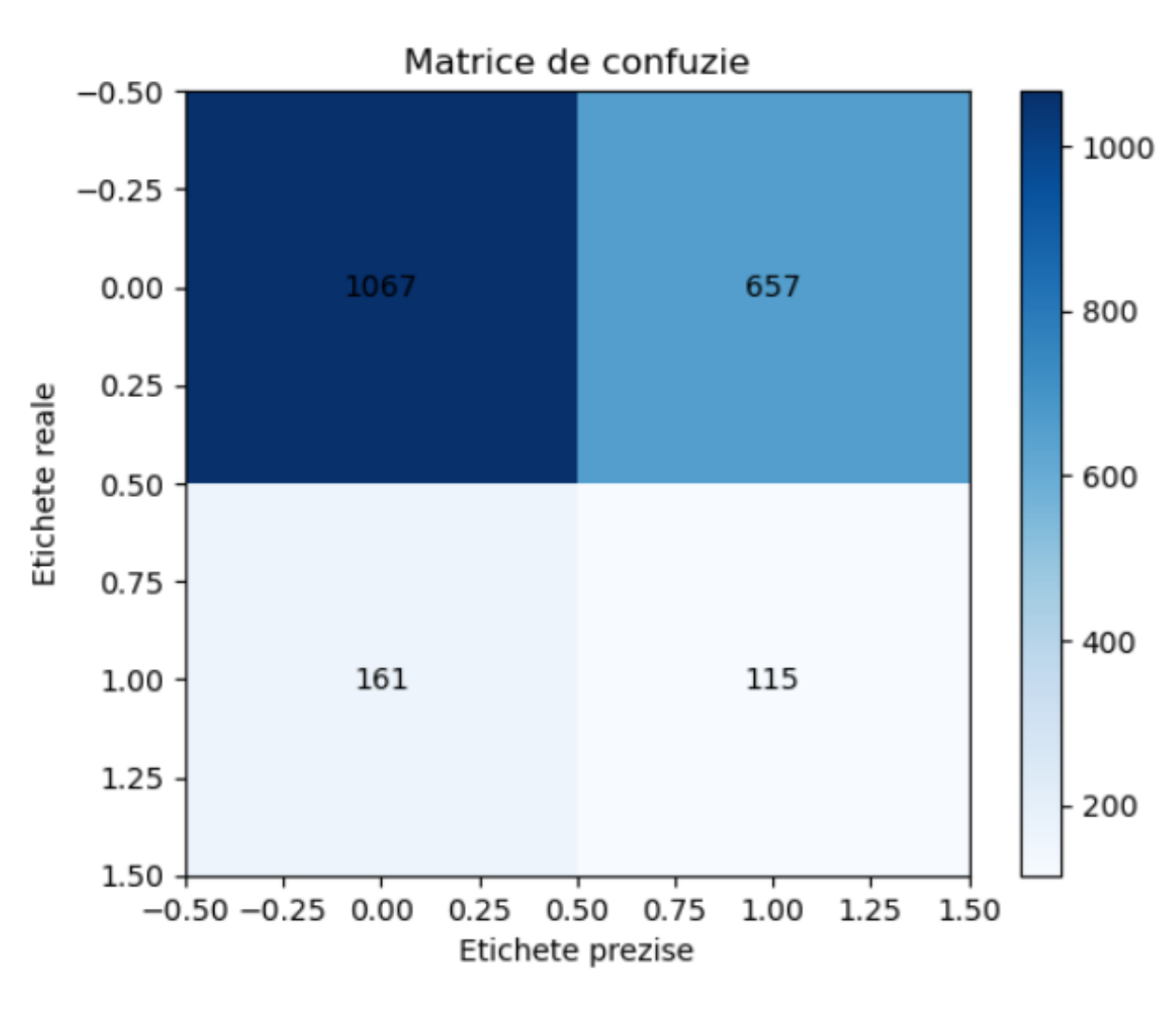
3.1.2 Hyperparameter tuning:

Paremetrul pe care am dorit sa il tunez este numarul de intervale in care se delimiteaza pixelii. Astfel, am decis sa testez pentru aceste numere:*{3,4,5,6,7,8}*.

Pentru imagini **128 x 128**:

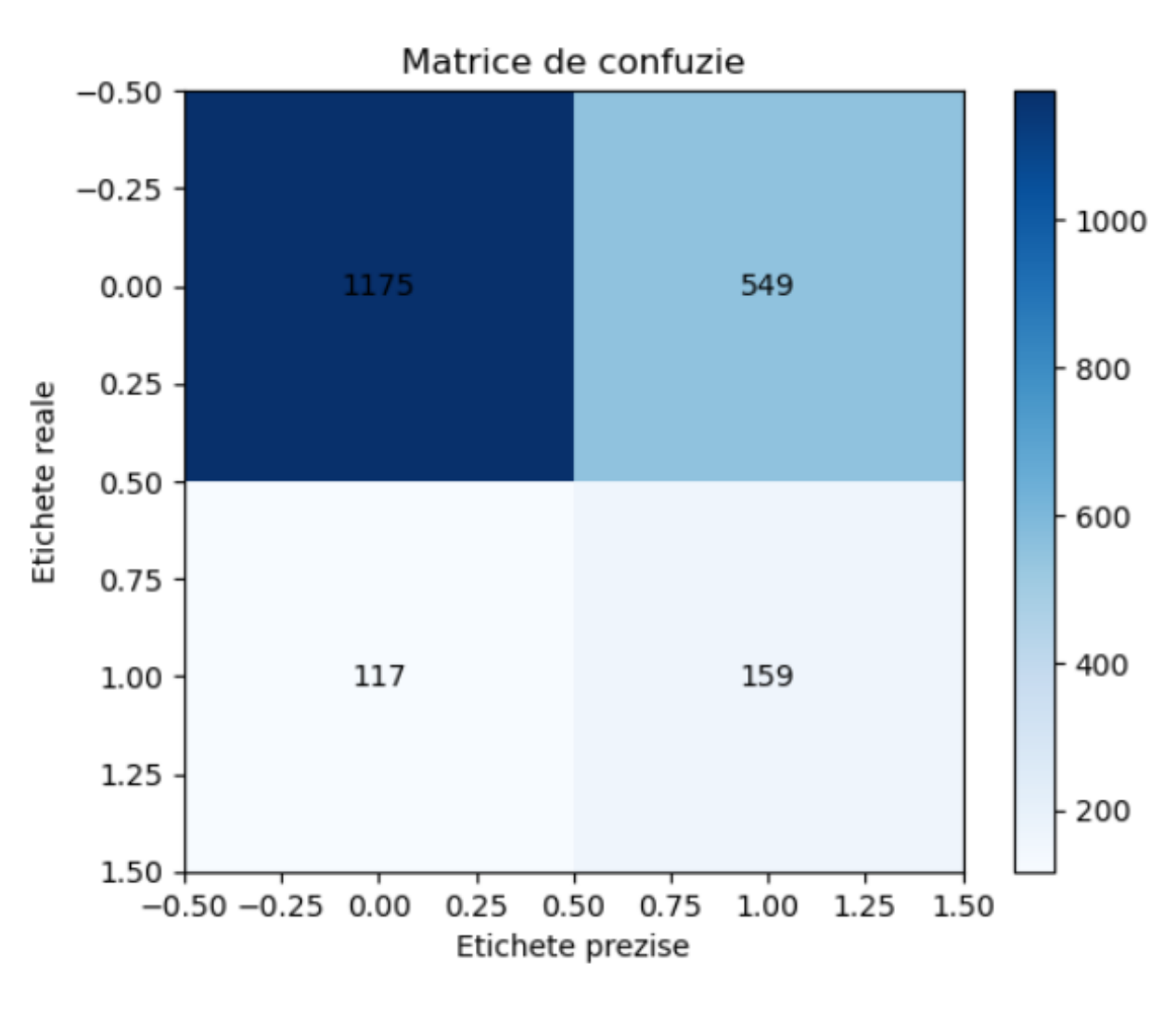
Intervale	Accuracy	Precision	Recall	F1 score
3	0.5905	0.1487	0.416	0.2192
4	0.591	0.1489	0.416	0.2194
5	0.5135	0.1476	0.5289	0.2308
6	0.5135	0.1476	0.5289	0.2308
7	0.514	0.1477	0.5289	0.2310
8	0.5155	0.1475	0.5253	0.2303

Matrice de confuzie pentru împărțirea pixelilor în 4 intervale:



Pentru imagini **128 x 128 GaussianBlur + CLAHE**:

Intervale	Accuracy	Precision	Recall	F1 score
3	0.667	0.2245	0.5760	0.3231
4	0.667	0.2245	0.5760	0.3231
5	0.667	0.2245	0.5760	0.3231
6	0.667	0.2245	0.5760	0.3231
7	0.667	0.2245	0.5760	0.3231
8	0.667	0.2245	0.5760	0.3231



3.2 Rețele neuronale convoluționale

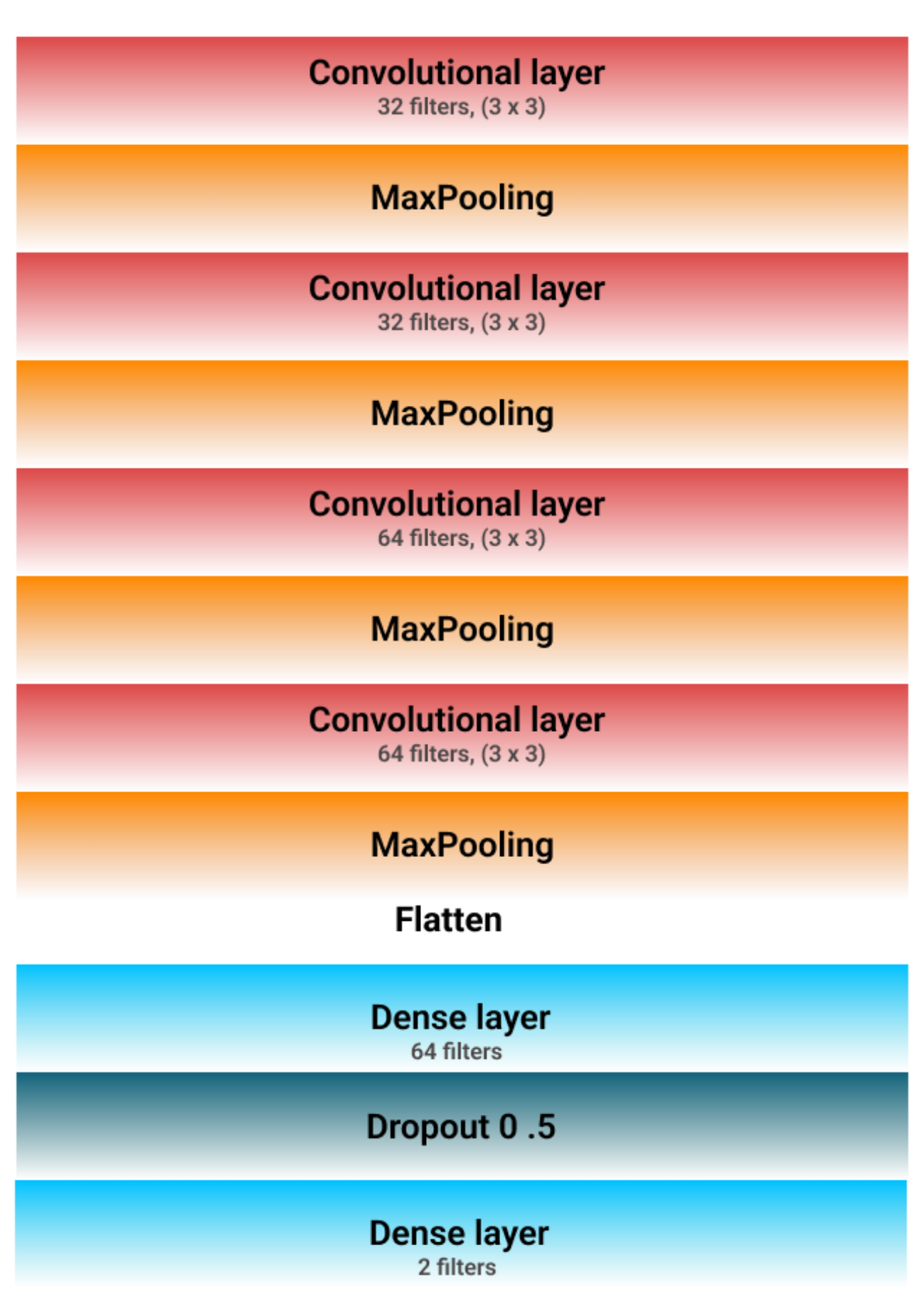
Următoarea abordare pe care am ales-o au fost **rețelele neuronale convoluționale**, deoarece sunt recunoscute pentru eficacitatea și acuratețea cu care pot construi un clasificator care analizează imagini.

3.2.1. CNNv1

Întâi am plecat de la o arhitectură simplă, în care sunt **4 straturi convoluționale**, fiecare urmat de un strat de **MaxPooling**. Primele două straturi au *32 de filtre*, matricea kernel de mărime *(3 x 3)*, următoarele două având *64 de filtre* și matrice identică .

Funcția de activare pentru straturile convoluționale este **ReLU**, funcție care transmite mai departe input-ul dacă este pozitiv, iar dacă nu este transmite 0.

După cele 4 straturi convoluționale, am adăugat un strat **Flatten** pentru a transforma ultimul output într-un vector 1 dimensional. În continuare, am adăugat două straturi conectate în întregime, **Dense**. Între straturile Dense am aplicat **Dropout** cu valoarea 0.5, strat ce va normaliza rețeaua prin ignorarea anumitor neuroni cu probabilitatea de 0.5.



Pentru funcția de pierdere, am folosit **BCEWithLogitsLoss** din PyTorch, care combină *activarea sigmoid* cu o funcție binară de pierde *cross-entropy*. Activarea sigmoid este aplicată output-ului din ultimul layer al rețelei(cel Dense). Optimizatorul ales pentru aceasta rețea este **Adam**, cu learning rate-ul egal cu 0.001.

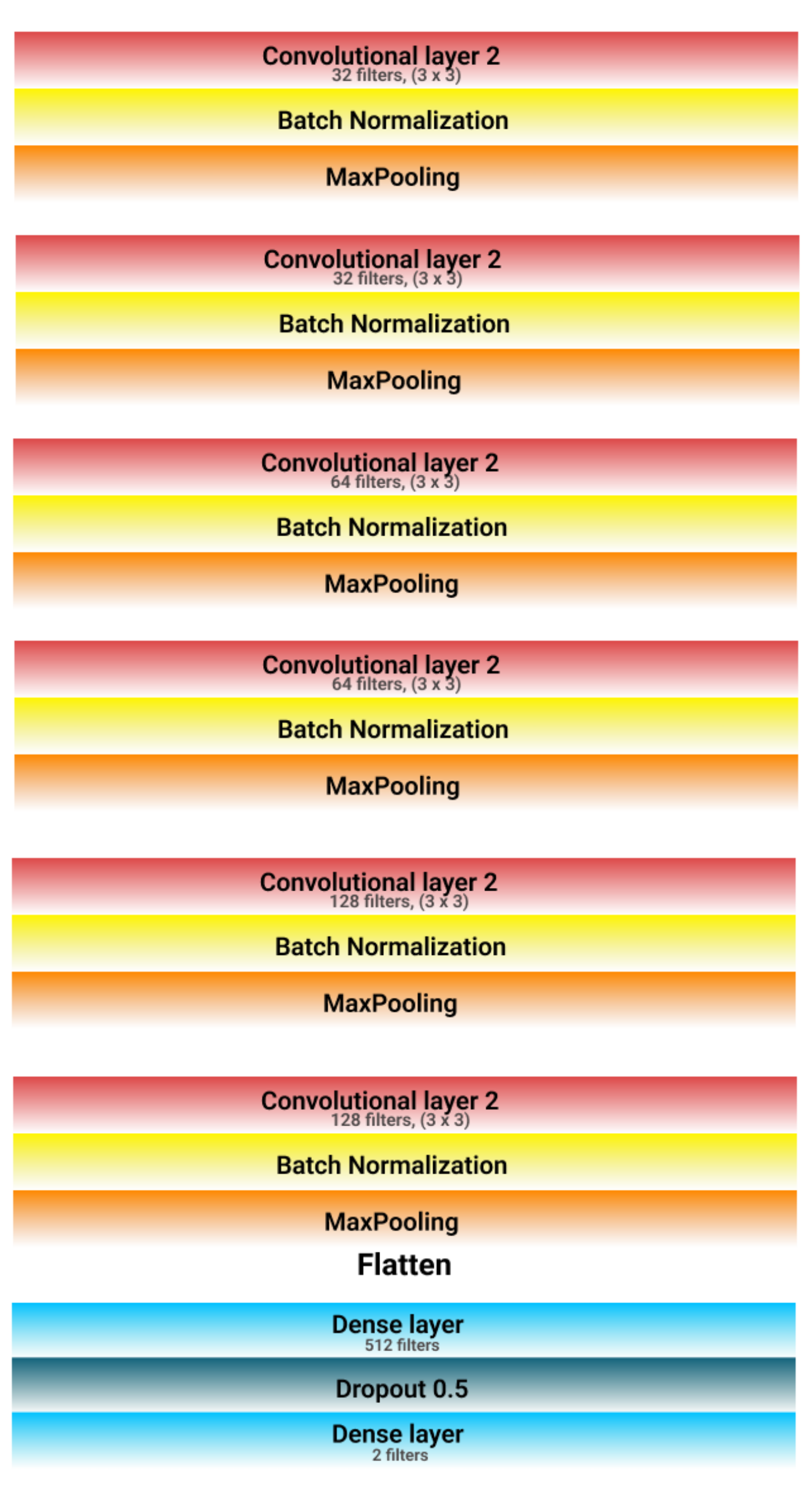
Am antrenat modelul pentru **20 de epoci** și am obținut următoarele rezultate pentru datele de validare:

	0.000	10.000	20.000
Accuracy	0.876	0.879	0.8795
Precision	0.632	0.618	0.590
Recall	0.243	0.322	0.413
F1 score	0.350	0.423	0.486

Întâi, pentru setul de date uzual, am obținut o acuratețe de **87.6%**, și un F1 score de **35.0%**. Odată cu adăugarea de imagini augmentate, atât acuratețea și F1 score-ul au crescut. De aceea am decis să măresc rețeaua.

3.2.2 CNNv2

În următorul model, pentru a evita un potențial overfit din cauza măririi numărului de filtre, cât și numărul straturilor convoluționale în sine, am adăugat pentru fiecare strat **BatchNormalizaion**. Scopul acestui strat este de a normaliza și mai mult rețeaua, calculând media și varianța input-urilor fiecărui strat, urmând că input-urile să fie modificate încât să aibă *media 0* și *varianța 1*.



Am antrenat modelul pentru **20 de epoci** și am obținut următoarele rezultate pentru datele de validare:

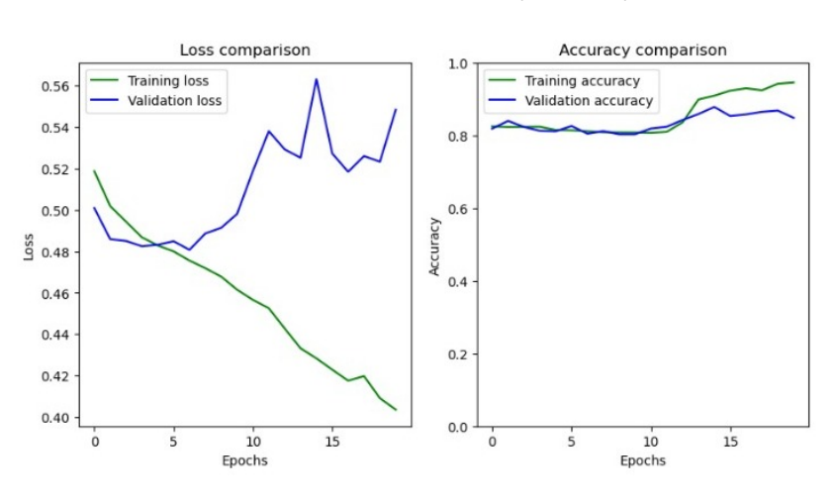


Fig 1. No augmented images

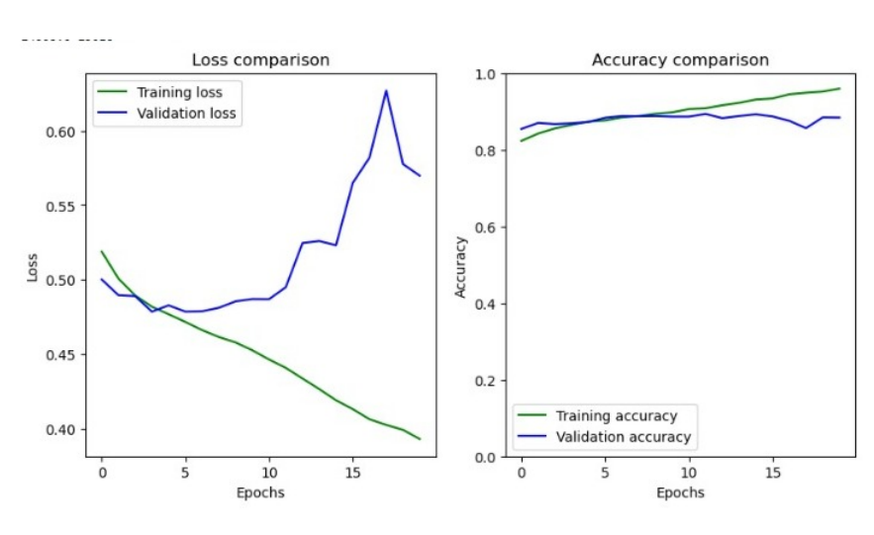


Fig 2. 10.000 augmented images

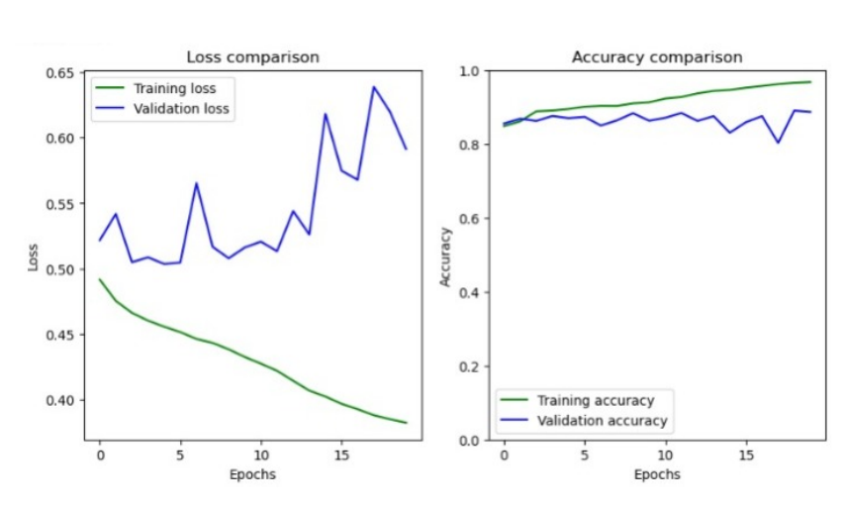


Fig 3. 20.000 augmented images

	0.000	10.000	20.000
Accuracy	0.8985	0.903	0.881
Precision	0.693	0.759	0.565
Recall	0.475	0.435	0.609
F1 score	0.563	0.553	0.586

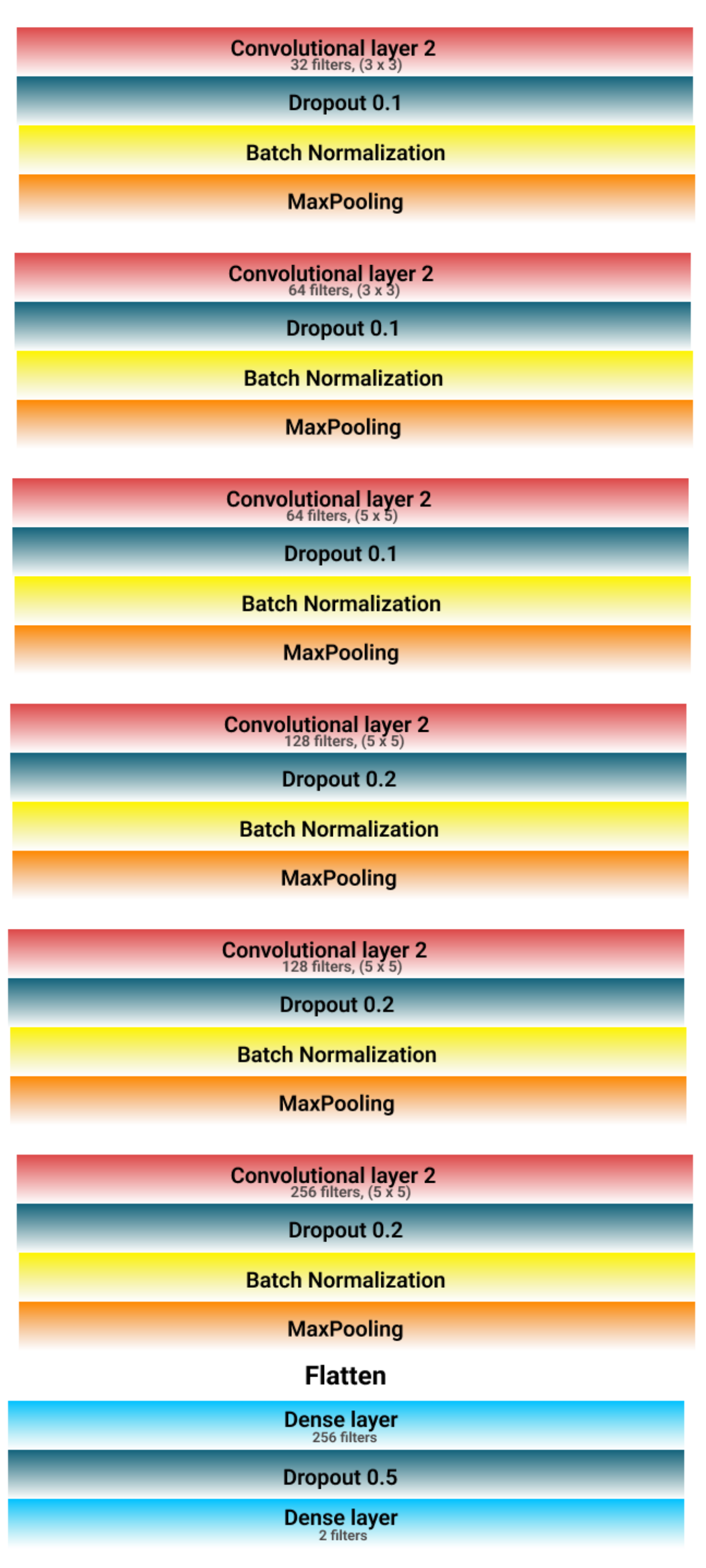
În ciuda faptului că odată cu creșterea numărului de imagini augmentate acuratețea a rămas ridicată și scorul F1 a crescut, pe grafice se poate observă creșterea pierderilor în cadrul validării, fapt ce indică că modelul face overfitting pe datele de antrenare. Nu am antrenat pentru mai multe epoci, deoarece tendința de creșțe a funcției de pierdere, cât și scăderea acurateții sunt evidente.

În continuare, am căutat cum să normalizez și mai mult modelul acesta.

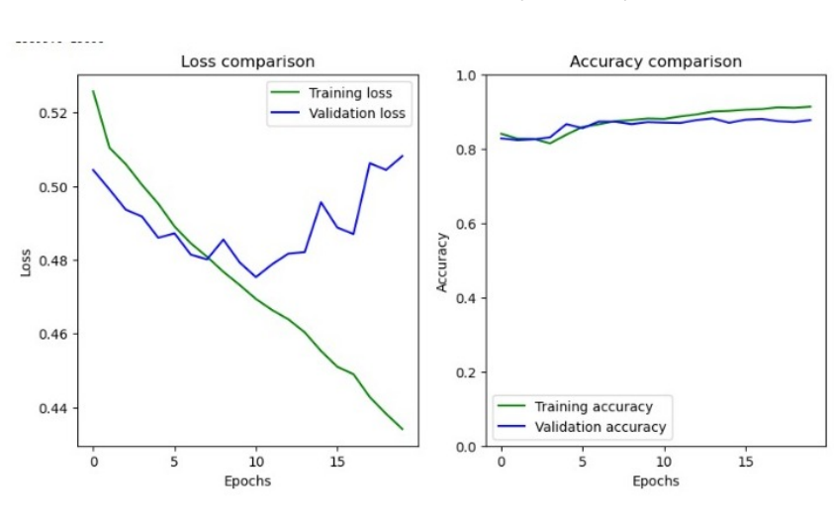
3.2.3 CNNv3

Pentru a ameliora fenomenul de overfit, am introdus după fiecare strat convolutional un strat de **Dropout**, cu probabilitatea de a renunța la un neuron egală cu 0.1. Această idee mi-a venit în urmă citirii acestei lucrări: Analysis on the Dropout Effect in Convolutional Neural Networks, by Sungheon Park and Nojun Kwak, în care este analizat efectul unui Dropout cu valoare mică în rețelele neuronale convoluțtonale.

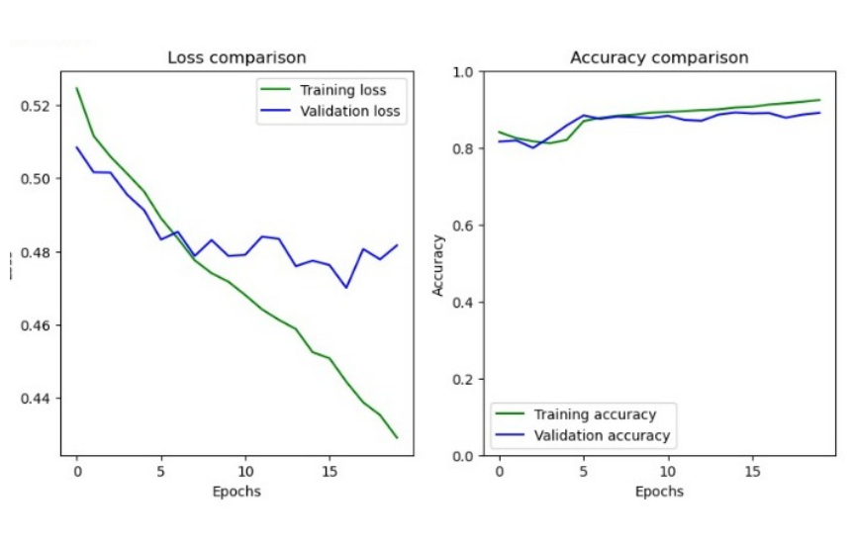
De asemenea, am mărit numărul de filtre pentru ultimele straturi convoluționale, cât și mărimea matricii kernel din (3 x 3) în (5 x 5).



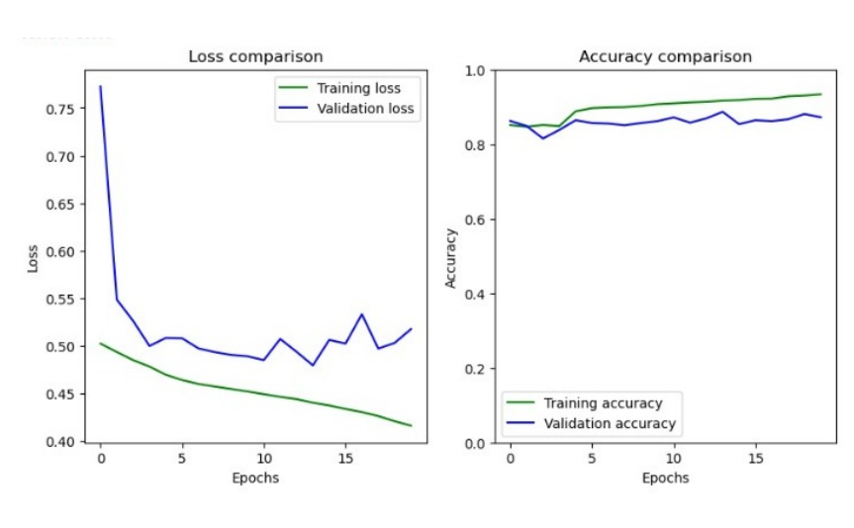
Am antrenat modelul pentru **20 de epoci** și am obținut următoarele rezultate pentru datele de validare:



No augmented images



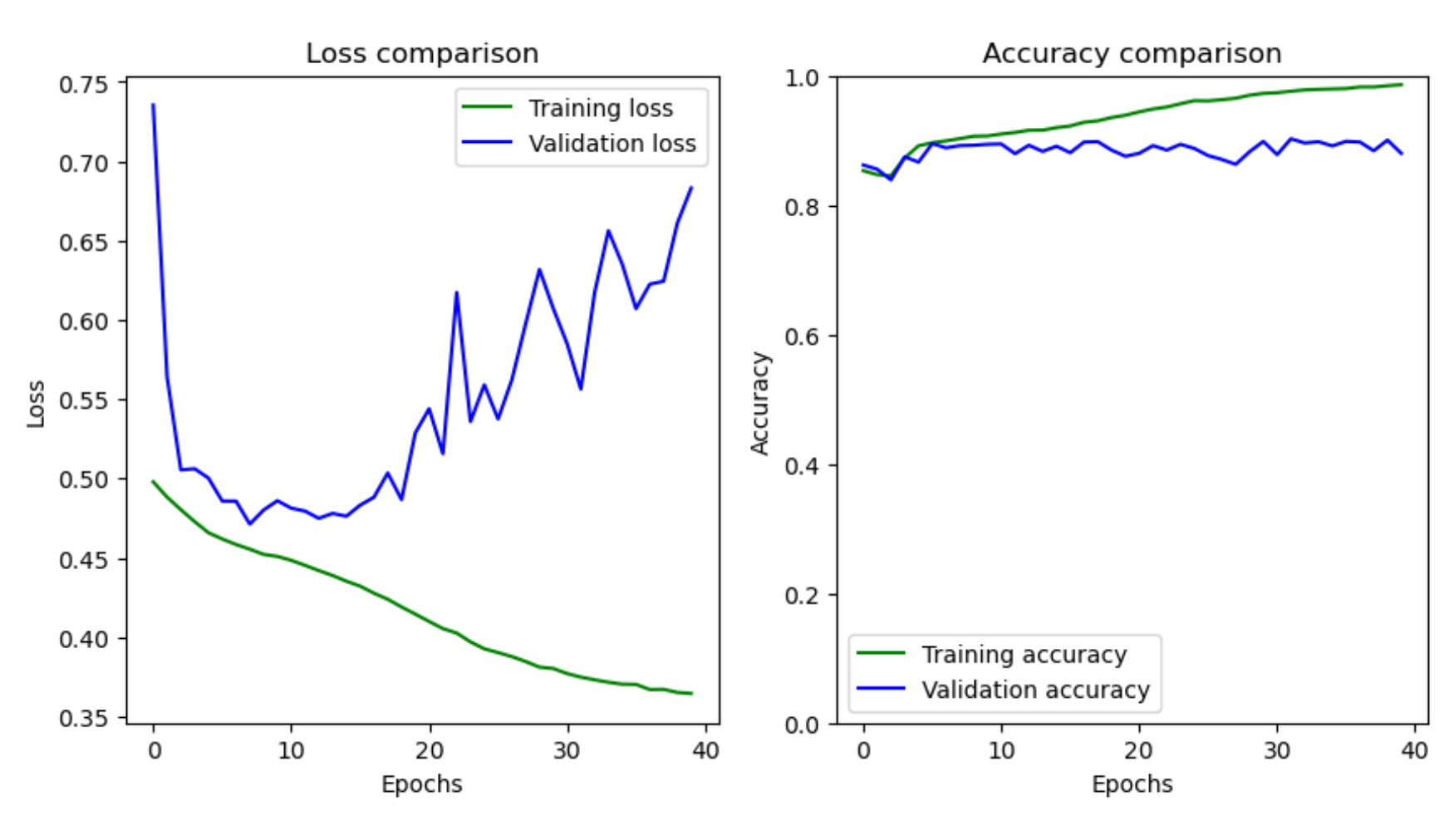
10.000 augmented images



20.000 augmented images

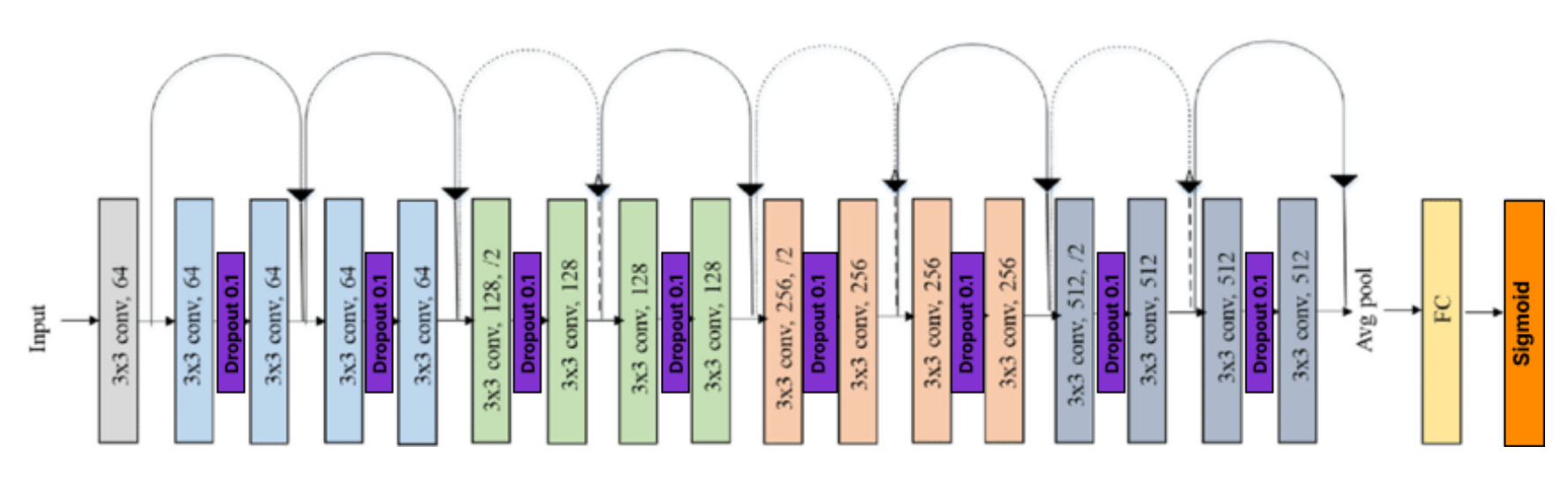
	0.000	10.000	20.000
Accuracy	0.897	0.8965	0.887
Precision	0.824	0.8	0.901
Recall	0.322	0.333	0.199
F1 score	0.464	0.470	0.326

Odată cu creșterea numărului de imagini augmentate, funcția de pierdere pentru validare s-a apropriat și mai mult de funcția de pierdere pentru testarea. În urmă acestei constatări, am decis să antrenez încă o dată acest model pentru **40 de epoci**.



4. Modelul final

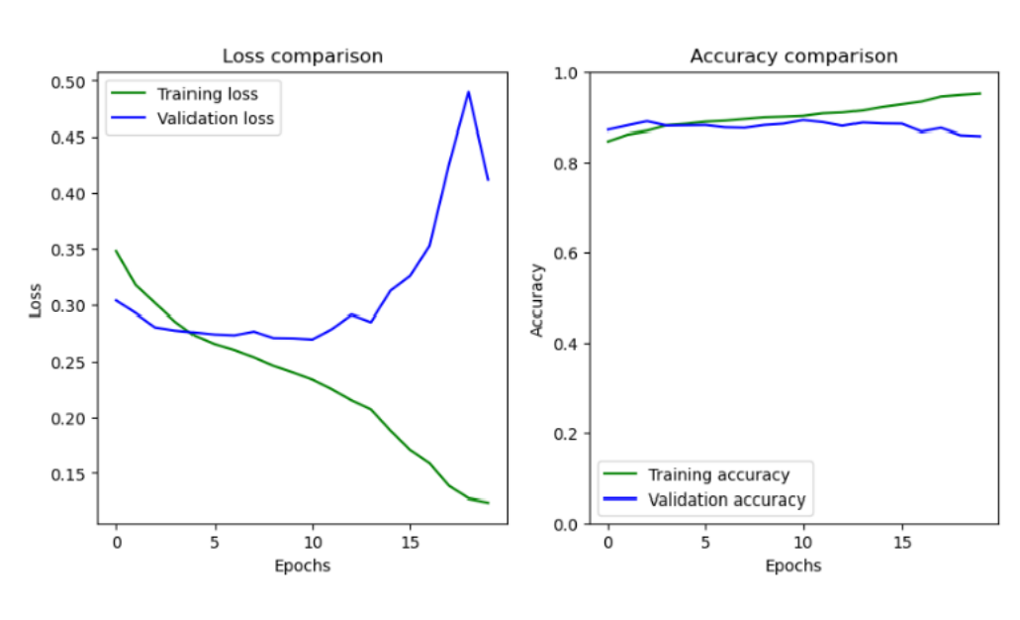
Modelul final pe care l-am incercat si pe care l-am ales pentru aceasta competitie a fost ResNet18. Pentru primele incercari am folosit in continuare optimizatorul **Adam** cu learning rate-ul 0.001, ca si pentru celelalte modele de pana acum.



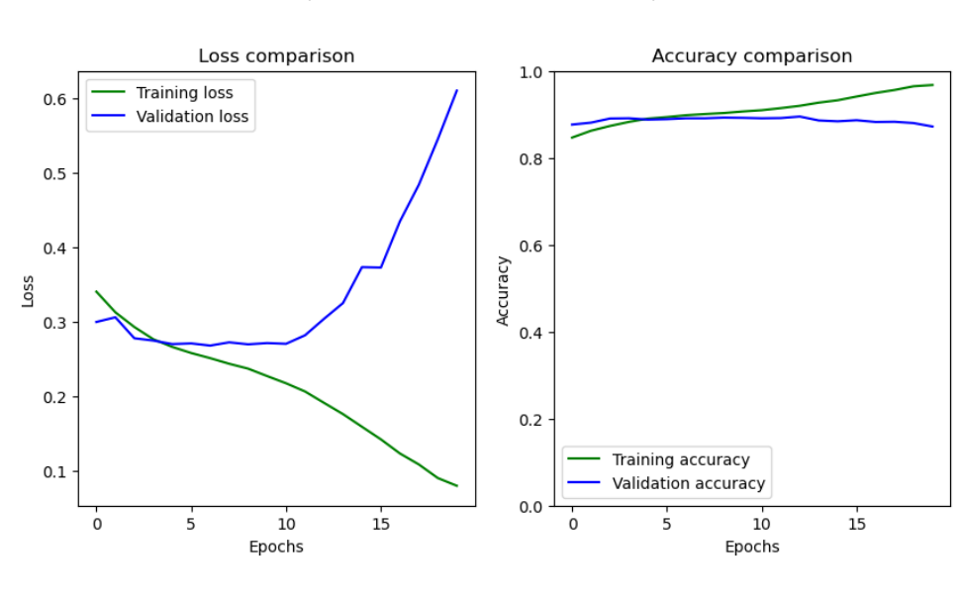
Peste această arhitectură bine cunoscută, am adăugat un strat de Dropout cu probabilitatea egală cu 0.1, imediat după primul strat convolutional din fiecare grupare de straturi.

Exemplu: După primul strat din prima grupare de straturi albastre(3x3 conv. 64) am adăugat Dropot(0.1), analog pentru celelalte grupări.

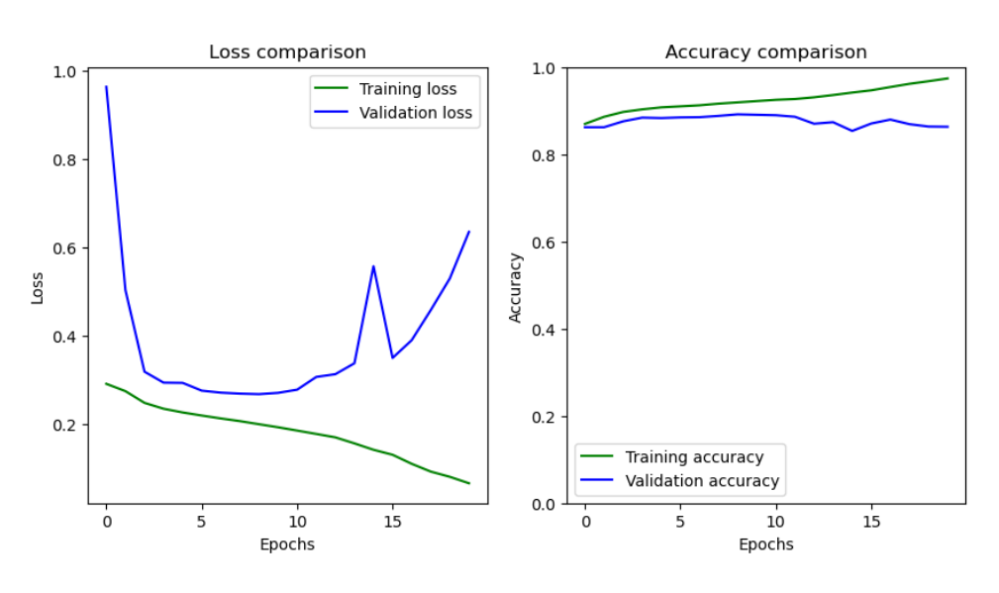
Am luat această decizie deoarece arhitectură fiind complexă, eu nu puteam augmenta mai mult de 25000 de imagini, din motive computaționale, iar normalizarea adițională ar putea compensa această lipsă.



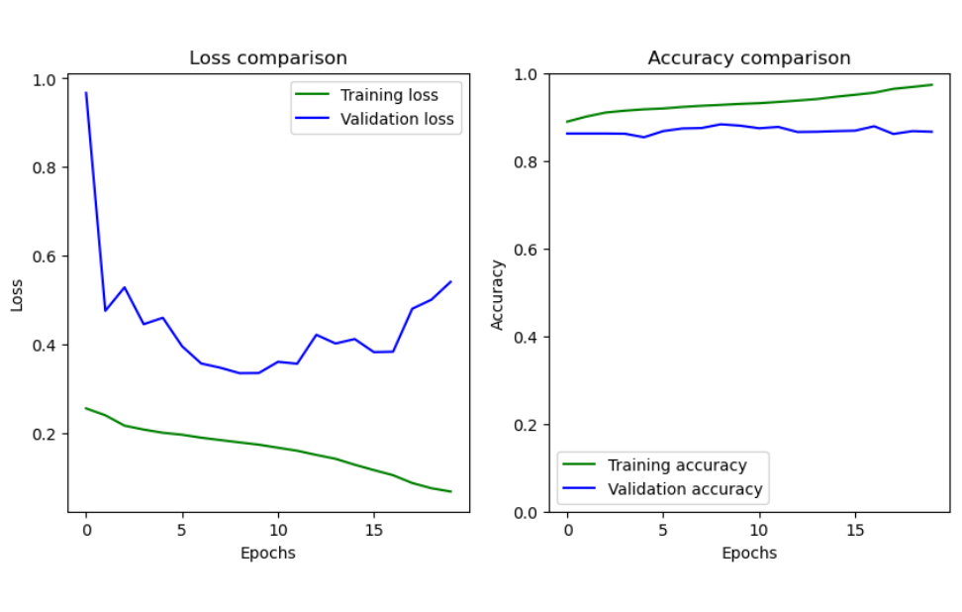
No augmented images



10.000 augmented images



20.000 augmented images



25.000 augmented images

	0.000	10.000	20.000	25.000
Accuracy	0.891	0.8735	0.884	0.878
Precision	0.6464	0.5497	0.6428	0.5625
Recall	0.4637	0.4601	0.3586	0.5217
F1 score	0.540	0.5009	0.4604	0.5413

5. Concluzii

În concluzie, dacă aș fi continuat dezvoltarea modelului, m-aș fi axat asupra normalizării imagnilor mai mult, să aflu cum este analizată de către un medic o astfel de tomografie, pentru a modela imaginile corespunzător. De asemenea, pentru rețelele neuronale convoluționale aș fi încerca altfel de algoritmi de optimizare și aș augmenta și mai multe imagini.