Inteligență Artificială - Machine Learning

CT Scan Classification

Nicolae Ducal
Grupa 234

Mai 20, 2021

Cuprins

CT Scan Classification

1.1 Descrierea problemei

„This is a CT scan classification challenge in which competitors have to train machine learning models on a data set containing computer tomography (CT) scans of lungs. Competitors are scored based on the classification accuracy on a given test set. For each test image, the participants have to predict its class label." (Sursă: Kaggle CT Scan Classification, seria 23)

1.2 Potențiale metode de rezolvare a problemei

Pentru că avem o problemă de clasificare, și anume antrenarea unui model eficient de clasificare a unor imagini, unele potențiale soluții care le-am putea folosi în primul rând ar fi:

- Clasificatorul Naive Bayes;
- Metoda celor mai apropiați K vecini (K-Nearest Neighbors);
- Clasificatorul cu Vectori Suport (SVM - Support Vector Machine).

Pentru soluții mai avansate, ar trebui să apelăm la rețelele neuronale, și anume la Rețelele Neuronale Convoluționale (Convolutional Neural Networks - CNN), care sunt specializate anume pentru task-ul de clasificare a imaginilor (la fel cum Rețelele Neuronale Recurente (RNN) sunt folosite pentru clasificarea audio).

Pentru clasificatoarele Naive Bayes și SVM putem să le folosim pe cele implementate în librăria Scikit-Learn. Pentru Kth - Nearest Neighbors putem folosi ideile de implementare de la laboratorul de ML. Pentru rețelele neuronale, cel mai eficient este să folosim librării specializate în Deep Learning, precum Pytorch și Tensorflow.

1.3 Soluția cu K - Nearest Neighbors

Prima soluție cu care am vrut să încerc să rezolv această problemă a fost metoda celor mai apropiați K vecini. KNN este o metodă eficientă pentru rezolvarea atât a problemelor de clasificare, cât și a celor de regresie. În cazul nostru, pentru că avem o problemă de clasificare, algoritmul funcționează în felul următor:

1. Se decide asupra parametrilor K, care va reprezenta numărul de vecini pe care îi luăm în considerare și o funcție de distanță, care ia ca parametri două date de tipul celor din seturile de train și test și returnează distanța respectivă dintre ele
2. Se inițializează fiecare element din mulțimea de train cu clasa sa

3. Pentru fiecare element din mulțimea de test, se determină K cei mai apropiați vecini ai elementului respectiv utilizând distanța definită, și i se atribuie clasa majoritară din cele K clase (în caz de egalitate, se aplică anumite departajări, iar pentru a diminua probabilitatea de numar egal de clase majoritate, de obicei numărul K se alege un număr impar)

Pentru acest model, m-am inspirat din soluția de la laboratorul de IA/ML și am implementat o soluție asemănătoare pentru această problemă.

Parametrii modelului: Acuratețea modelului KNN depinde în mare parte de doi parametri: K - numărul de vecini pe care vrem să-l luăm în considerare când facem clasificarea datelor de validare și test și metrica cu care vrem să calculăm distanța dintre două elemente. În cele mai multe cazuri, parametrul K se alege un număr impar.

Etapele realizării:

1. În primul rând, am încercat câteva variante banale: K = 7, Metrica 'L2', K = 5, Metrica 'L1', etc. Am observat că obțin o acuratețe în range-ul $\approx 35 - \approx 55$. Mi-am dat seama că un astfel de rezultat nu este foarte bun și am trecut la o altă abordare.

2. Pentru a realiza o oarecare analiză mai bună, am făcut unele schimbări în program: pentru K făceam o căutare într-o listă de date introduse de mine (1, 2, 3, 4, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 101, 103, 105, 201, 203, 205, 1001), pentru metrici am adăugat o funcție nouă (Minkovski) care calculează distanța Minkovski de ordinul n, iar ca metrici am ales să fie calculate metricile 'L1', 'L2', 'L3', 'L4'. Din cauza faptului că numărul de operații era unul foarte mare (Nr. de K * Nr. de Metrici * Operatii clasificare), dacă rulam totul pe CPU, s-ar fi executat mai încet totul. Din această cauză, am luat și am trecut de la array-uri din Numpy la tensori din Pytorch, pe care mai apoi i-am mutat pe GPU (menționez că am folosit GPU-urile folosite de Google Colab). Astfel, am putut să calculez într-un timp foarte rapid toate rezultatele cu diferiți parametri, obținând o statistică care nu m-a impresionat nici acum. În cazul acestui model, nu am folosit nici un fel de preprocesare a imaginilor. Am observat că odată cu creșterea K-ului, crește și acuratețea modelului pe datele de validare, dar nu cu mult față de cele din prima etapă. De asemenea, am observat o creștere a acurateții când treceam de la distanța Manhattan la distanța Euclidiană, scăzând apoi la calcularea distanțelor Minkovski de ordin 3 și 4. Cea mai bun rezultat l-am obținut cu K= 205 și distanța Euclidiană: ≈ 57.5 pe datele de validare. O reprezentare mai bună a distribuției se poate observa în acest grafic:

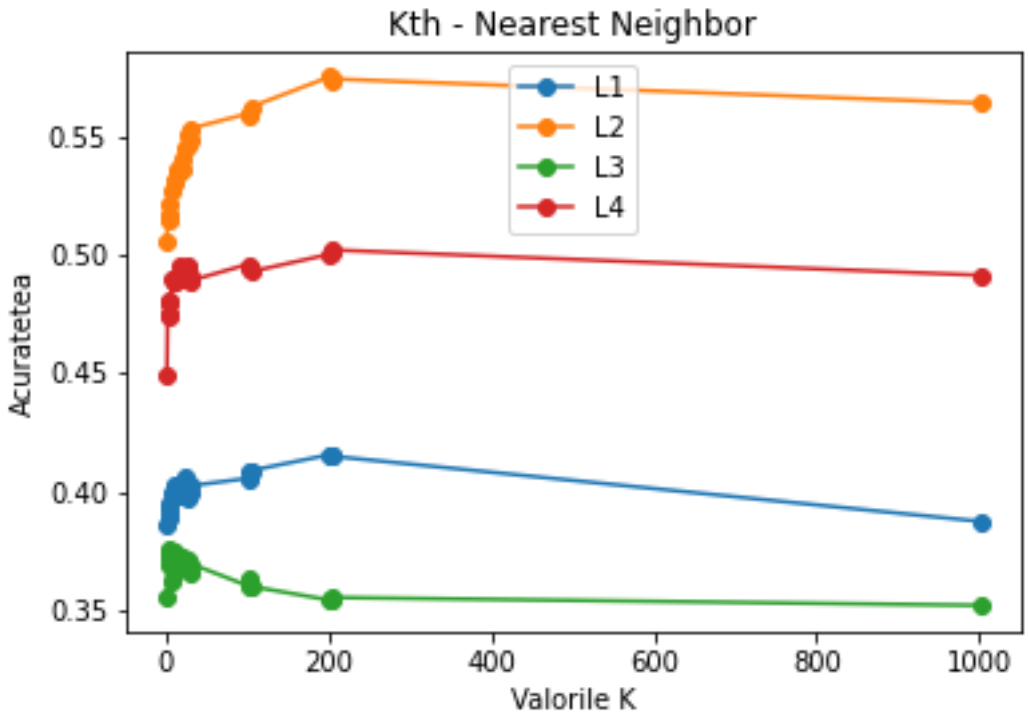

Figure 1.1: Graficul acurateții modelului de KNN în dependență de parametrii acestuia

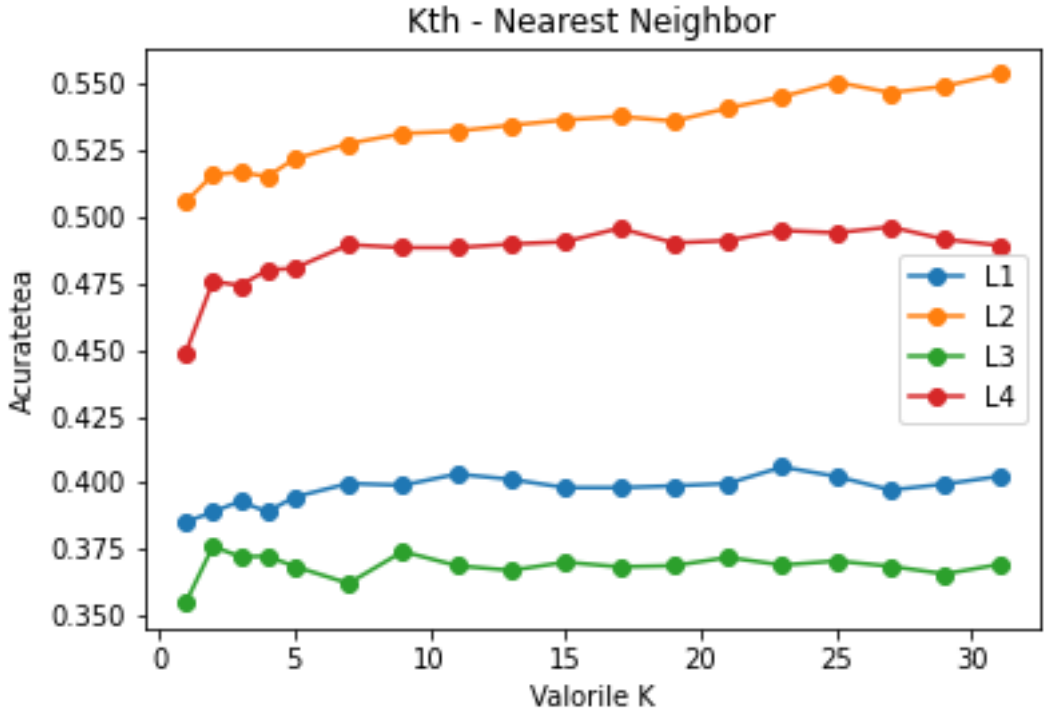

Figure 1.2: Graficul acurateții modelului de KNN (cu K până la 50)

De asemenea, matricea de confuzie pentru modelul cu cea mai bună acuratețe (K=205 și metrica L2) este prezentat în următorul tabel:

Observam ca distribuția nu este foarte bună: de exemplu peste 700 de imagini din clasele 1 și 2 au fost misclasificate ca clasa 0, 422 imagini din clasa 1 au fost misclasificate ca clasa 2 etc.

	Clasa 0	**Clasa 1**	**Clasa 2**
Clasa 0	1133	166	201
Clasa 1	395	683	422
Clasa 2	341	389	770

Din cauza faptului că nu puteam obține valori foarte mari ale acurateții pe datele de validare, nu am încercat să fac submit acestei soluții și am hotărât să trec deodată la ceva mai avansat, precum o rețea neuronală convoluțională (CNN).

1.4 Modelele de CNN

Rețea neuronală convoluțională: Rețelele neuronale convoluționale sunt o varinată specială de rețele neuronale, care au fost concepute în special pentru a clasifica mai bine imagini. Acestea au început să fie utilizate în ultimii ani mai pe larg, odată cu triumful arhitecturii AlexNet la concursul de clasificare de imagini ImageNet. Deosebirea dintre CNN și o Rețea Neuronală Fully Connected (FCNN) este introducerea a unor noi straturi, pe lângă cele dense (fully connected): de convoluție, pooling, normalizare etc., rolul principal avându-l primele și operația de convoluție pe baza căreia funcționază acestea. În fiecarui strat de convoluție se definește un filtru (de obicei de dimensiuni mici și desigur, mai mici ca dimensiunile inputului), prin care trece imaginea, efectuând la fiecare pas operația de convoluție (suma produselor dintre elementele filtrului si imaginii). Pentru o explicație mai simplă și clară a operației de convoluție, acesta poate fi interpretată ca operația de identificare a unor semne, pattern-uri într-o imagine. Operația de pooling (de obicei se folosesc max și average) presupune selectarea celor mai mari valori dintr-o porțiune a imaginii (în cazul max) sau a mediei valorilor din acea porțiune (în cazul average). Normalizarea batch face o normalizare a valorilor după layerul în care a fost amplasat pe fiecare channel. De obicei, acestea sunt introduse în cazul CNN după straturile de convoluție. Alte plusuri ale CNN-urilor ar fi că procesarea imaginilor prin straturile de convoluție, iar la final introducerea a unor layeruri fully-connected este mult mai bună decât dacă am lua și am face deodată liniarizarea a unei imagini de dimensiune $N * M$, pentru că am pierde toate particularitățile imaginii și am face o rețea foarte mare care nu s-ar antrena destul de bine.

Librăriile și modulele utilizate: În cadrul competiții am folosit atât librăria Tensorflow (Keras), cât și Pytorch. Într-un final, am decis să antrenez majoritatea modelelor cu librăria Pytorch, pentru simplitatea de lucru și posibilitatea de definire a unor funcții după placul utilizatorului.

Preprocesarea imaginilor și datelor: Citirea pozelor din toate seturile de date a fost efectuata în modul RGB (cu librăria Pillow). Astfel, inițial, toate pozele aveau dimensiunea de (3, 50, 50). Pentru preprocesarea imaginilor, am folosit unele metode de transformare a pozelor din Pytorch, modulul torchvision.transforms. În primul rând, imaginile inițiale au dimensiunea de 50x50, ceea ce este destul de puțin. Am observat că odată cu mărirea dimensiunii imaginilor, acuratețea modelelor folosite a crescut considerabil (unele chiar cu $7 - 10\%$). Într-un final, am decis să folosesc un resize până la dimensiunea de 270, apoi să fac un center crop de dimensiune 256, pentru a selecta mai bine detaliile dintr-o imagine. O eventuală mărire a imaginilor cel mai probabil ar fi îmbunătățit rezultatele, însă din cauza limitei de memorie pentru un GPU în Google Colab, nu puteam defini imaginile cu dimensiuni mai mari de 500 și să le folosesc pe modelele utilizate de către mine. Apoi, am aplicat unele transformări doar pe datele de train, pentru a oferi puțină diversitate imaginilor și posibilitatea ca modelul să învețe mai bine particularitățile claselor:

1. Rotații aleatoare ale imaginilor cu n grade, n fiind ales din intervalul [5, 25];

2. Flip-uri aleatoare orizontale ale imaginilor, cu probabilitatea în intervalul [0.05, 0.1];

În final, pentru toate seturile de date am folosit transformarea ToTensor(), pentru a face conversia imaginilor în tensori și am adaugat o normalizare a datelor, cu parametri sugerați de librărie.

Pentru citirea datelor de train, validare și test, am creat un dataset custom (DatasetCTScanKaggleNicuDucal), care ne permite să creăm dataseturi pentru fiecare categorie. Beneficiul dataseturilor este faptul că putem crea cu ajutorul lor DataLoader-e, care automat împart pozele în batch-uri de dimensiune introdusă de către utilizator, permite amestecarea datelor, pentru a nu antrena un model de fiecare dată pe date în aceeași ordine și a face overfit rapid pe datele din train. Pentru dimensiunea batch-urilor, în general, se folosesc puteri ale lui 2. În cele mai multe cazuri am utilizat batch-uri de dimensiune 32, dar am încercat și cu 8, 64 și 128. Opțiunea de amestecare a datelor (shuffle) am activat-o doar pentru datele de train.

Alegerea parametrilor: Principalii parametri de care depinde modelul de CNN ar fi rata de învățare (learning rate) a modelului, optimizatorul care folosește acest parametru și funcția de pierderi (loss function). Pentru funcția de pierderi am folosit una din cele mai utilizate funcții pentru CNN - CrossEntropyLoss. Pentru optimizator am folosit mai multe variante, precum:

- SDG (Stochastic Gradient Descent) cu parametri default
- Adadelta
- Adagrad
- Adam

Cel mai eficient s-a dovedit a fi Adam, care, de fapt, combină proprietățile altor optimizatoare, precum Adagrad. Pentru learning rate am utilzat valori între 10^{-5} și 10^{-4}, cel mai des folosind chiar aceste valori.

Modelele utilizate: În cadrul acestei competiții, am încercat mai multe tipuri de rețele neuronale convoluționale: atât modele definite de mine, cât și arhitecturi predefinite și preantrenate din librăria Pytorch.

1. **CNN definit de mine:** Prima rețea neuronală a fost definită de către mine și avea mai multe variante. De exemplu, una dintre ele avea următoarele caracteristici: 3 straturi de convoluție, cu câte 3 straturi de MaxPooling după fiecare din ele și activație ReLU. După acestea urmează 3 starturi fully-connected, la fel cu activație ReLU. Această variantă folosită obținea în practică un rezultat destul de bun. Primele două submisii au fost efectuate cu această variantă și au obținut 65%, respectiv 71%.

2. **Arhitectura AlexNet, definită de mine:** A doua rețea neuronală a fost o imitare a rețelei neuronale AlexNet, adaptată pentru pozele cu dimensiunea de (3, 50, 50), din motiv că la această etapă nu am folosit resize-urile imaginilor. O caracteristică a rețelei AlexNet este faptul că numărul de canale crește brusc la un numere destul de mari (de exemplu: primul strat crește de la 3 la 96, apoi la 256, etc.), iar dimensiunile imaginii scad considerabil. Cu această variantă am obținut soluții de 72% și 73%. O potențială aplicare a resize-ului imaginilor probabil că ar fi mărit cu câteva unități rezultatul.

3. **VGG:** A doua arhitectură de CNN utilizată au fost cele din seria VGG. Dintre toate variantele posibile, am folosit versiunile VGG16 și VGG19. VGG este o arhitectură eficientă și bine cunoscută, fiind una din primele rețele neuronale convoluționale care a introdus pe larg utilizarea unui număr foarte mare de straturi de convoluție, normalizare batch, pooling-uri (max și average). VGG se caracterizează prin faptul că folosește un număr mare de layeruri convoluționale, iar straturile de convoluție sunt de dimensiunea 3x3, cu padding 1, care nu modifică dimensiunea imaginii. Am optat pentru aceste două variente din următoarele motive:

(a) VGG16: Este o rețea relativ mai mică decât succesorii ei (are mai puține straturi decât VGG19) și nu are problemă foarte mare cu vanishing gradients.

(b) VGG19: Este cea mai mare rețea de tip VGG și ar avea un potențial înalt în soluționarea problemei (o problemă mică fiind numărul mare de straturi de convoluție fără să fie completate cu skip connection sau alte tehnici, care ar putea genera un mic vanishing gradients).

Am încercat atât o implementare de mână, modificată în unele cazuri de mine, cât și versiunea predefinită și preantrenată pe baza de date a ImageNet-ului, testând cu diferite rate de învățare, dimensiunea imaginilor. Pe VGG-urile definite de către mine am obținut valori de 76-78% pe datele de validare (cu image size-ul de 50x50), 83-84% (cu image size-ul de 256x256). Cu VGG19 preantrenat am obținut valori de 88-89% pe datele de validare, dar $\approx 82 - 83\%$ pe datele de test.

1.5 DenseNet - Arhitectura câștigătoare

DenseNet: Cea mai mare acuratețe și, în special, cea cu care am obținut soluția finală de $\approx 88\%$ pe datele de test am obținut-o cu ajutorul modelului predefinit și preantrenat (la fel ca și celelalte arhitecturi, pe baza de date ImageNet) - DenseNet. DenseNet este un upgrade al predecesorilor săi, cum ar fi ResNet, VGG, care se caracterizează prin mai multe particularități:

- **Skip Connections:** Problema micșorării gradientului (vanishing gradient) la propagarea backward a acestuia poate apărea nu doar când folosim funcții de activare precum sigmoida, tanh, care pot ușor da valori foarte aproape de 0, dar și atunci când rețeaua devine foarte adâncă. Astfel, în aceste cazuri, valorile gradientului în straturile superioare devine foarte mică, rețeaua ajungând să nu se antreneze. Pentru a soluționa această problemă a fost introdusă noțiunea de skip connection, care practic unește un layer ar rețelei de alte layer-uri precedente (cu excepția predecesorul său, cu care evident va fi unit). Dacă înainte, funcția unui layer era definită de formula $Layer(x_i)$, acum aceasta va avea forma $Layer(x_i, x_j, \dots)$, în dependență de numărul de skip connection-uri pe care le are layer-ul curent. Noțiunea de skip connection a apărut în arhitectura ResNet. Astfel, folosind această metodă, vanishing gradient-ul ajunge să fie o problemă mică. Ce este caracteristic anume pentru DenseNet (față de ResNet) este numărul foarte mare de conexiuni de acest tip, care apar aproape între o bună parte din layer-uri din blocurile Dense.

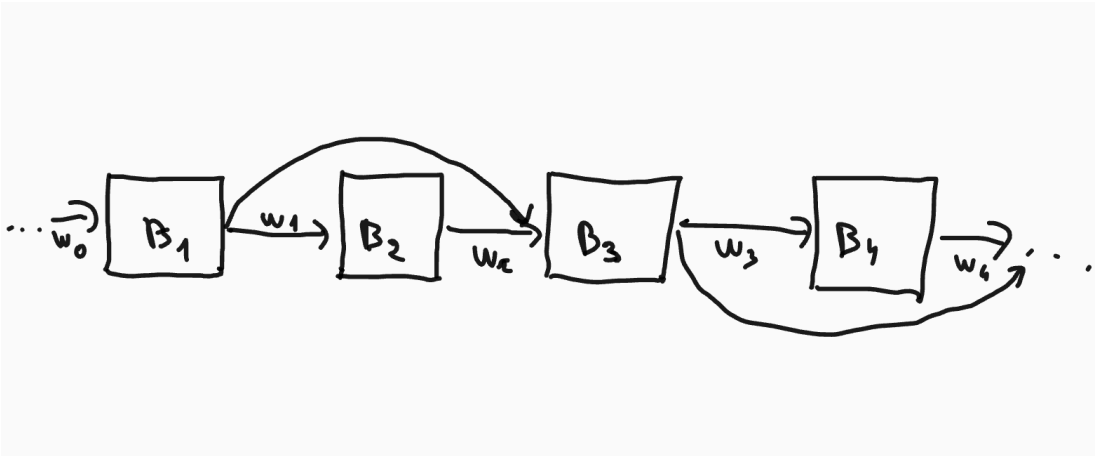

Figure 1.3: Reprezentare a skip connection-urilor

- **DenseBlocks și DenseLayers sau "moștenirea" particularităților:** O altă caracteristică importantă a arhitecturii DenseNet sunt blocurile dense (DenseBlocks). Fiecare DenseBlock este compus din câteva layer-uri dense (DenseLayers), aceastea fiind în principal compuse din:

  1. Batch Normalizaton + ReLU + Convoluție 2D (kernel_size de 1) - acestă parte din layer este folosită în special pentru mărirea numărului de canale pe care le au datele de input
  2. Batch Normalizaton + ReLU + Convoluție 2D (kernel_size de 3) - partea respectivă este utilizată pentru extragerea particularităților imaginii

  Un DenseBlock este constituit dintr-un număr anumit de DenseLayer-uri și funcționează un felul următor: la început, blocul, respectiv primul DenseLayer, primește niște features de o anumită dimensiune și un anumit număr de canale. Particularitatea DenseNet-ului este că fiecare DenseLayer următor va primi nu doar features-urile procesate de layer-ul precedent, ci de toate layer-urile dinaintea acestuia. Acest lucru are loc datorită utilizării Skip Connection-urilor. Astfel, fiecare DenseLayer are o informație mult mai multă, având posibilitatea să extrage unele particularități/features pe care nu au reușit să le scoată layer-urile precedente. O vizualizare a funcționării și structurii unui DenseBlock poate fi vizualizată în următoarele imagini:

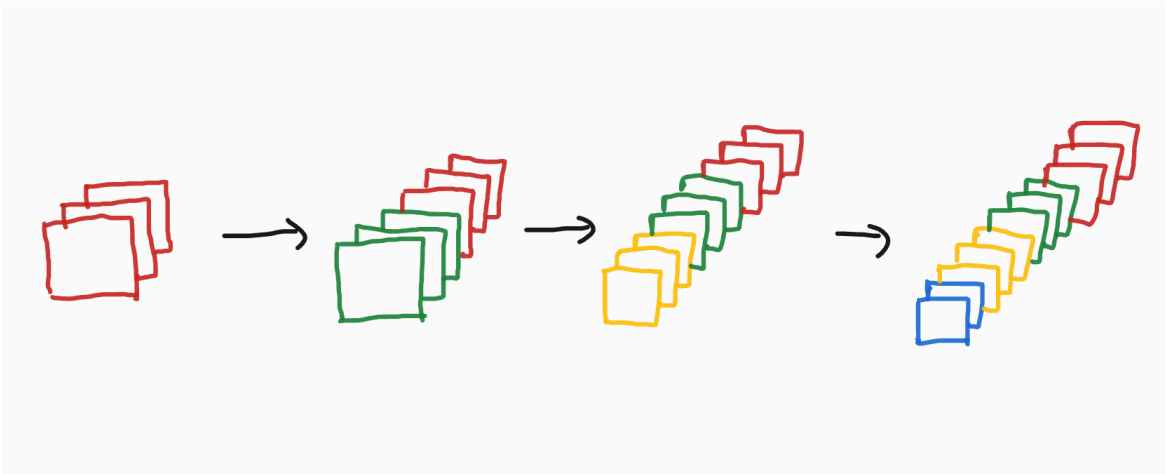

Figure 1.4: Reprezentarea principiului de funcționare a unui DenseBlock (fiecare strat reprezintă inputul unui DenseLayer)

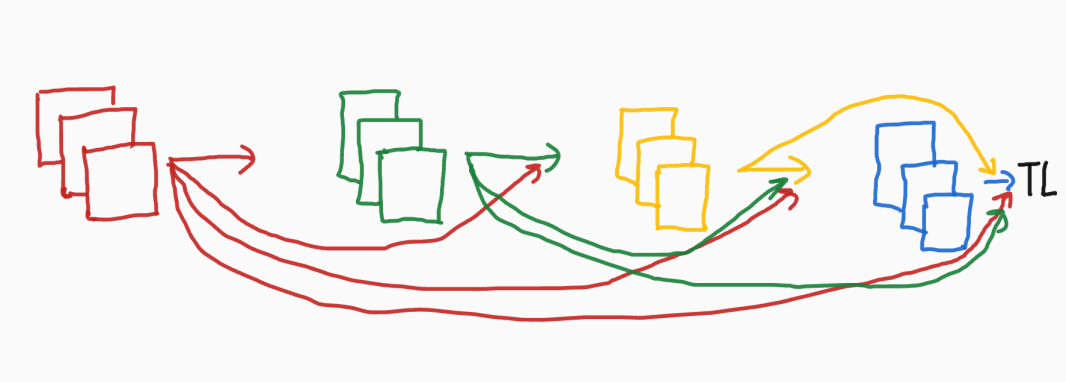

Figure 1.5: Reprezentarea structurii a unui DenseBlock

- **Transition Layers:** Altă particularitate specifică DenseNet-ului sunt layer-ele de tranziție (Transition Layers) dintre DenseBlocks. Acestea au fost introduse cu scopul ca dimensiunea

features-urilor la intrarea în fiecare block să fie identică. Aceste layer-uri sunt constituite dintr-un Batch Normalization, o activare ReLU, un strat de convoluție cu kernel de dimensiune 1 și un pooling.

Rezultate: DenseNet are mai multe versiuni (precum DenseNet 121, 161, 169, 201), în dependență de mărimea rețelei. Cel mai bun rezultat l-am obținut cu DenseNet 161 și următorii parametri: rata de învățare $= 10^{-5}$ și optimizatorul Adam. Din start am observat că optimizatorul Adam a funcționat mult mai bine decât ceilalți trei (statistică prezentată în tabelele ce urmează). De asemenea, a fost interesant să constat faptul că pentru learning rate-ul de 10^{-4}, modelul se antrena ma rapid, dar nu se ridicau mai sus de 90-91% pe datele de validare. Aceeași remarcă am făcut-o și în cazul cu rata de 10^{-5}, doar că antrenarea a decurs puțin mai lent inițial, dar validarea la fel s-a oprit la 89-91%. Am fost plăcut surprins să văd că de fapt, soluția cea mai bună antrenată cu learning rate-ul mai mic a obținut un scor mai mare, într-un final, pe datele de test, cele cu learning rate de 10^{-4} având un scor de $\approx 84 - 85\%$ pe datele de test. Probabil, unul din motive ar fi că modelul ar fi fost antrenat prea mult, fiind overfit. În următorul tabel voi reda matricea de confuzie pentru cel mai bun rezultat obținut:

	Clasa 0	**Clasa 1**	**Clasa 2**
Clasa 0	1463	21	16
Clasa 1	14	1157	329
Clasa 2	3	50	1447

Table 1.1: Matricea de confuzie pentru cel mai bun rezultat

Analizând această matrice, observăm că imaginile din clasa 0, la fel ca și în cazul VGG-ului, au fost clasificate relativ bine. De asemenea, modelul recunoaște mult mai bine imaginile din clasa 2, însă, din păcate, a misclasificat peste 300 de imagini din clasa 1 ca clasa 2. Acest lucru reflectă faptul că, cel mai probabil, pozele din clasa 1 conțin anumite particularități din clasa 2, modelul întâlnind dificultăți la clasificarea acestora.

În continuare, voi prezenta graficele acurateții și pierderilor pentru acest model:

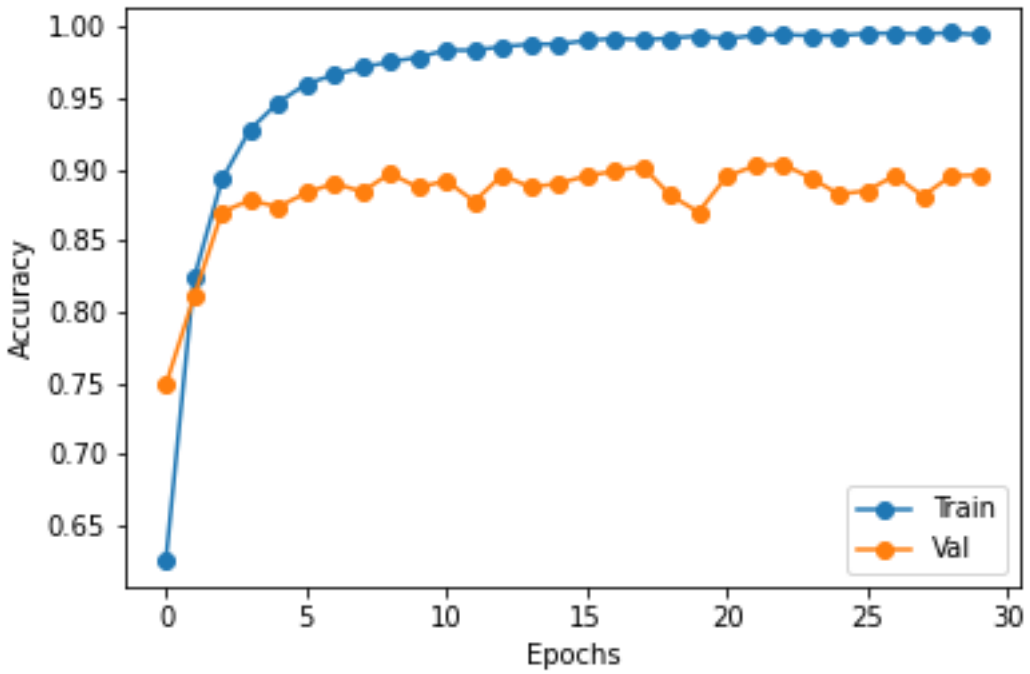

Figure 1.6: Graficul acurateții modelului preantrenat DenseNet în dependeță de epocă (learning rate 10^{-5})

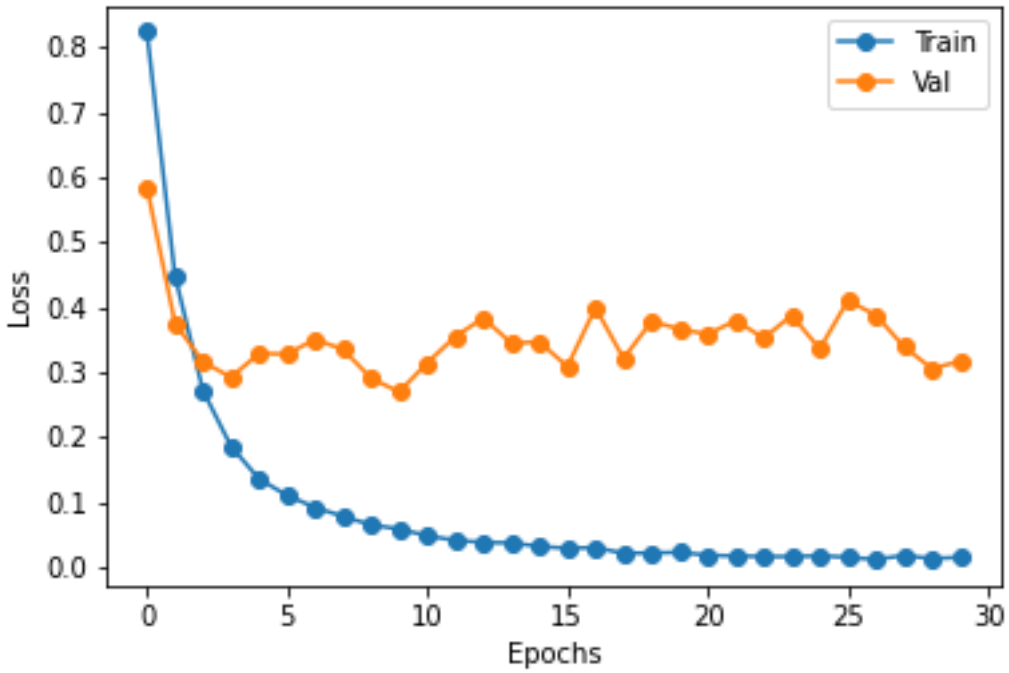

Figure 1.7: Graficul loss-ului modelului preantrenat DenseNet în dependeță de epocă (learning rate 10^{-5})

În final, voi prezenta statisticile pentru diferiți parametri cu acest model (doar pentru 20 de epoci):

1. **Learning rate -** 10^{-4} **și Optimizatorul Adam**:

	Acuratețe Train	**Acuratețe Validare**
Epoca 1	0.7914	0.87622
Epoca 5	0.97046	0.89222
Epoca 10	0.9792	0.88044
Epoca 15	0.98693	0.90688
Epoca 20	0.99333	0.87822

2. **Learning rate -** 10^{-5} **și Optimizatorul Adam**:

	Acuratețe Train	**Acuratețe Validare**
Epoca 1	0.6252	0.74888
Epoca 5	0.94706	0.87377
Epoca 10	0.97893	0.88733
Epoca 15	0.98813	0.89022
Epoca 20	0.99393	0.87022

3. **Learning rate -** 10^{-4} **și Optimizatorul Adagrad**:

	Acuratețe Train	**Acuratețe Validare**
Epoca 1	0.66733	0.76044
Epoca 5	0.8942	0.84622
Epoca 10	0.93586	0.86622
Epoca 15	0.95873	0.874
Epoca 20	0.9654	0.86133

4. **Learning rate -** 10^{-4} **și Optimizatorul Adadelta**:

	Acuratețe Train	**Acuratețe Validare**
Epoca 1	0.3294	0.30555
Epoca 5	0.39026	0.31733
Epoca 10	0.48466	0.40244
Epoca 15	0.52386	0.45822
Epoca 20	0.54693	0.50711

5. **Learning rate -** 10^{-4} **și Optimizatorul SGD**:

	Acuratețe Train	**Acuratețe Validare**
Epoca 1	0.35339	0.33466
Epoca 5	0.50613	0.52844
Epoca 10	0.5798	0.59644
Epoca 15	0.65493	0.65355
Epoca 20	0.70273	0.69599

Observații: Dintre toate optimizatoarele, observăm că cel mai bine se descurcă optimizatorul Adam (am folosit doar learning rate de 10^{-4} pentru a determina care e mai eficient, pentru că rezultatele ar fi fost asemănătoare dacă era 10^{-5}). Probabil dacă ar fi rulat mai multe epoci, optimizatoarele SGD și Adadelta ar fi oferit un rezultat mai bun, însă aceasta ar consuma mult timp și multe resurse, pentru că rețeaua este una foarte mare și densă. Cum am ales learning rate-ul de 10^{-5} am explicat în paragrafele de mai sus.

1.6 Concluzii

În cadrul aceestei teme, am avut ocazia să învățat și să folosim diferite metode de clasificare ale imaginilor, precum KNN, CNN și diferite arhitecturi de CNN. Introducerea în Rețele Neuronale și învățare Deep ne deschide calea spre tehnici avansate de programare și învățare din Machine Learning. Rezultatul obținut de mine în urma competiției este unul foarte bun, însă statistica și clasamentul demonstrează că există și alte modele și arhitecturi care garantează un rezultat mai precis decât atât.