Universitatea POLITEHNICA din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Proiect de semestru

Clasificarea scenelor terestre din imagini de teledetecție folosind ResNet

Student
Alex-Costin Țițu

Anul 2024

Cuprins

Lista Figurilor

Lista Tabelelor

Sumar: Beneficiind de tehnici pentru transferul cunoștințelor și metode de augmentare al seturilor de date, lucrarea își propune implementarea unei rețele ResNet în vederea clasificării scenelor terestre captate în imagini optice satelitare. Setul de date pus la dispoziție de către Universitate Merced din California urmărește clasificarea acestor scene, predominant urbane, regăsite în imagini optice satelitare înregistrate în spectrul vizibil. Setul de date împarte scenele regăsite în aceste imagini de teledetecție în 21 de clase frecvent întâlnite. Folosind capacitatea rețelelor neurale convoluționale adânci de a învăța operația de extragere a trăsăturilor definitorii în vederea separării imaginilor în clasele aferente, se va urmări implementarea unui model care să poată generaliza sarcina descrierii și înțelegerii scenelor satelitare folosind un număr cât mai mic de parametri. Rezultatele finale obținute vor fi comparate cu performanțele obținute de către metodele de ultimă oră (state-of-the-art) aplicate pe acest set de date.

1. Fundamente Teoretice

1.1. Modelul Deep Learning

Învățarea profundă (en. Deep Learning) este un subset al Inteligenței Computaționale, specific focalizat pe utilizarea rețelelor neuronale cu capacitatea de extragere în mod automat a trăsăturilor utile și a formelor (en. patterns) incluse în interiorul datelor. Trăsăturile extrase din datele neprelucrate sunt utilizate pentru a compune o sarcină de învățare informată și a lua decizii viitoare pe baza cunoștințelor acumulate. Pentru a informa deciziile, mai întâi este necesară învățarea modului de extragere al trăsăturilor din datele de intrare, trăsături care vor determina cum trebuie îndeplinită cu succes sarcina algoritmului.

Algoritmii tradiționali de învățare automată (en. Machine Learning), de obicei operează prin definirea unui set de reguli sau caracteristici în mediul datelor. Aceste reguli sunt de obicei definite manual de către un operator uman care va analiza datele și va încerca să extragă trăsăturile principale ale datelor. Acest proces de explorare al datelor în vederea extragerii caracteristicilor va avea o influență semnificativă asupra performanțelor algoritmilor de învățare, ei depinzând în mod sensibil de reprezentarea datelor.

Algoritmii de Deep Learning urmăresc rezolvarea acestor probleme cauzate de extragerea în mod eficient a caracteristicilor relevante din setul de date, abordarea fiind una cu totul diferită. Cheia rezolvării cu succes a sarcinilor este extragerea și învățarea trăsăturilor direct din setul de date, într-un mod ierarhic. Această extragere a trăsăturilor într-un mod ierarhic presupune ca fiind dat un set de date, spre exemplu un set de date cu sarcina de identificare a fețelor, se pune problema implementării unui model de rețea neurală care să primească ca dată de intrare o imagine, iar pe baza acesteia să detecteze diferite trăsături definitorii ale unei fețe. Primul pas este detectarea elementelor de bază, a contururilor (caracteristici de nivel jos), apoi construirea pe baza contururilor a nasului, ochilor și gurii (caracteristici de nivel mediu), iar cu ajutorul caracteristicilor de nivel mediu se vor compune elementele definitorii ale structurii faciale (caracteristici de nivel înalt). Astfel, pe măsură ce informația este procesată de straturile unei rețele neurale adânci, se va distinge abilitatea de a capta astfel de trăsături ierarhice, reprezentând avantajul învățării profunde comparativ cu învățarea automată clasică. Rețelele neurale adânci realizează extragerea caracteristicilor și reprezentarea lor dintr-un spațiu inițial complex, neprelucrat, într-un nou spațiu mai simplu, ce conține doar trăsăturile esențiale ale datelor de intrare.

Blocurile constructive fundamentale ale învățării profunde și algoritmii de bază există de decenii, însă domeniul a început să ia amploare de curând deoarece datele au început să fie din ce în ce mai răspândite. Datele sunt puterea motrice a foarte multor astfel de algoritmi iar azi ne uităm la o lume în care avem mai multe date ca oricând. Facilitatea colectării și stocării seturilor de date cât mai mari și mai diverse, de complexități ridicate, a condus la nașterea domeniului Big Data.

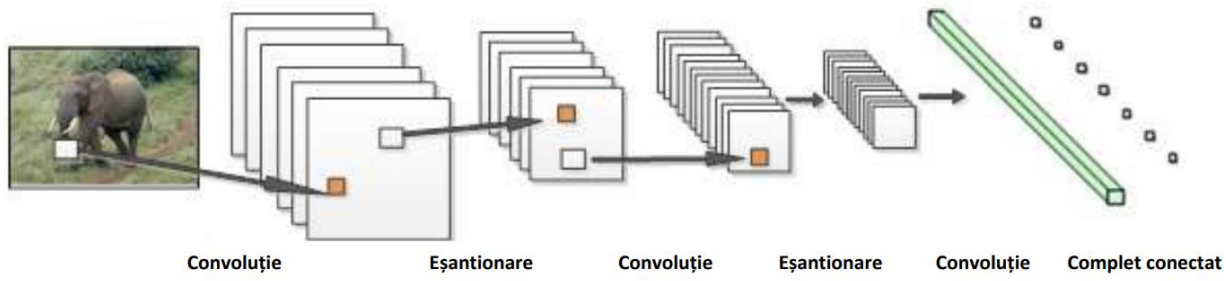
Un alt motiv pentru care domeniul a început de curând să capete atenție este că acești algoritmi și aceste modele de rețele neurale sunt masiv paralelizabile. Până acum puterea de calcul și arhitectura sistemelor de calcul (serială) nu era suficientă pentru a putea implementa aceste modele și acești algoritmi foarte adânci. Datorită avansurilor ce au avut loc în ultimele decenii în domeniul arhitecturilor multi-CPU și multi-GPU, dar mai ales în domeniul procesoarelor grafice, acum modelele neurale pot beneficia de avantajele majore aduse de noile tehnologii precum framework-ul NVIDIA Compute Unified Device Arhitecture (CUDA). CUDA este o platformă de calcul paralelă ce permite dezvoltatorilor să accelereze dramatic aplicațiile de calcul prin valorificare puterii arhitecturii paralele a procesoarelor grafice.

În ultimul rând, cu ajutorul uneltelor software open source precum PyTorch sau Tensorflow este posibilă construirea și aplicarea algoritmilor complecși de DL mult mai facil, iar prin utilizarea acestor biblioteci software alături de framework-ul CUDA timpul de antrenare și testare al rețelelor neuronale adânci este redus semnificativ.

1.2. Rețele neurale convoluționale

Rețelele neurale convoluționale (en. Convolutional Neural Networks – CNN) sunt modele neurale multistrat specializate în rezolvarea de probleme de detecție a formelor (en. pattern recognition), fiind dezvoltări inspirate din biologie ale modelului Perceptron Multistrat. Comparat cu rețelele prezentate anterior de tip feedforward cu un același număr de straturi, modelul CNN are mult mai puține conexiuni și, implicit, un număr de parametri mult mai redus. Acest lucru conduce la o antrenare mult mai simplă și totodată la performanțe superioare de recunoaștere.

CNN-urile sunt rețele neurale ierarhice ce se bazează pe operații de convoluție intercalate cu straturi de interpolare (eșantionare), având drept scop extragerea caracteristicilor esențiale precum și reducerea dimensiunii datelor. Operația finală de clasificare (sau regresie liniară) este realizată de un număr de straturi complet conectate (fully connected) introduse la finalul rețelei convoluționale, după extragerea trăsăturilor esențiale de nivel înalt.



Figură 1.1 Structură CNN [1]

1.2.1. Strat de convoluție

Stratul de convoluție este descris de numărul de hărți de caracteristici (en. feature maps) dat de numărul de nuclee (filtre) utilizate în acel strat, dimensiunea nucleelor (kernels) și conexiunile cu stratul precedent. Adițional se ține cont de modul în care are loc convoluția, dacă matricea este bordată și care este dimensiunea bordării (en. padding), alături de dimensiunea saltului ferestrei de convoluție (en. stride).

Fiecare strat conține un număr M de hărți de caracteristici cu aceeași dimensiune (M_x, M_y), fiecare hartă de caracteristici este un rezultat al sumei convoluțiilor hărților de caracteristici din stratul precedent cu nucleele lor corespondente și procesate de un filtru liniar. La final se adaugă un termen de bias (ro. pierdere) și se aplică o funcție de activare neliniară

$$M_{i,j}^k = ReLU\big((W^k * x)_{i,j} + b_k\big),$$

unde M^k este harta caracteristicilor cu index k, caracterizată prin ponderile W^k și termenul de bias b_k, folosind funcția ReLU.

Dimensiunile hărții de ieșire sunt:

$$M_x^k = \frac{M_x^{k-1} + 2P_x^k - K_x^k}{S_x^k} + 1$$

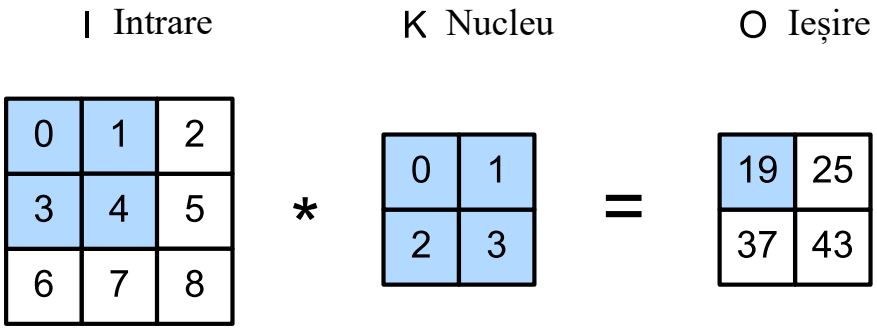
$$M_y^k = \frac{M_y^{k-1} + 2P_y^k - K_y^k}{S_y^k} + 1$$

unde indexul k indică stratul, K_x și K_y sunt dimensiunile nucleului, P_x și P_y sunt dimensiunea bordării, iar S_x și S_y sunt numărul de pixeli săriți de nucleu în procesul de convoluție denumiți stride sau skipping factors.

Convoluția unui semnal bidimensional discret este definită prin următoarea relație:

$$o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[u-m, v-n]$$

Neuronii dintr-o hartă de caracteristici își împart ponderile prin reutilizarea nucleelor în straturile convoluționale, acest fapt are ca efect creșterea eficienței prin reducerea numărului de parametrii antrenabili. Ca rezultat, CNN-urile vor avea un nivel de generalizare mai ridicat.



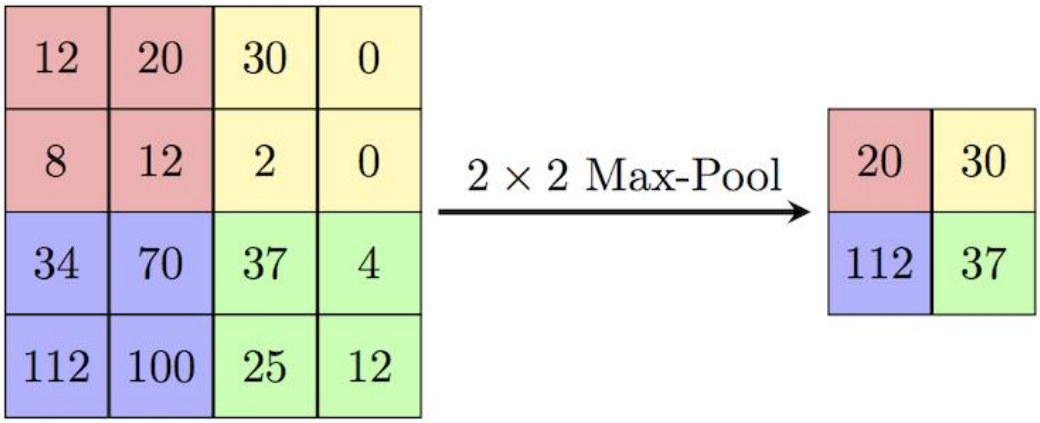
Figură 1.2 Operație de convoluție [2]

1.2.2. Strat de Pooling

Stratul de Pooling este un strat de eșantionare ce precedă stratul convoluțional. Cel mai frecvent utilizat tip de strat de Pooling este cel de tip Max-Pooling, ce reușește să scoată în evidență și să păstreze doar trăsăturile cele mai importante extrase în urma convoluției (ex: contururile).

Acest strat reduce dimensiunea hărților, îmbunătățind viteza de convergență (reduce efortul computațional) și crește capacitatea de generalizare datorită invarianței la poziție.

Ieșirea unui strat de Max-Pooling este dată de valoarea maximă de activare din regiunile non-overlapping (ro. non-suprapuse) ale nucleelor, venind ca o alternativă în locul abordării clasice de mediere a intrărilor, Average-Pooling. Un strat de Average-Pooling are ca ieșire media valorilor din interiorul regiunilor non-overlapping ale nucleelor.



Figură 1.3 Operație Max-Pooling cu fereastră 2x2 și 2 pixeli săriți [3]

1.2.3. Strat de clasificare/regresie

Se aleg parametrii modelului pentru straturile de convoluție, max-pooling și average-pooling astfel încât ieșirile ultimului strat convoluțional să formeze un vector unidimensional de caracteristici. O altă alternativă ar fi transformarea vectorului de trăsături rezultat la ieșirea ultimului strat convoluțional într-un vector unidimensional (en. flatten) și alegerea numărului de neuroni al primului strat complet conectat ca fiind egal cu dimensiunea vectorului unidimensional.

Acest vector de caracteristici va fi folosit ca intrare pentru o serie de straturi complet conectate (en. fully connected) care vor executa o operație de clasificare sau regresie liniară, în funcție de problema dată.

Ultimul strat din acest grup are câte un singur neuron pentru fiecare clasă sau câte un neuron pentru fiecare valoare continuă ce trebuie estimată prin regresie liniară.

2. Metoda abordată

În acest paragraf voi descrie metoda abordată pentru clasificarea scenelor urbane captate în imagini optice satelitare. Mai întâi va fi prezentat tipul de arhitectură al rețelei convoluționale utilizate, ResNet. Apoi, voi explica modul în care au fost decișii hiperparametrii optimi pentru antrenarea rețelei și cum am decis dacă este necesară sau nu o politică de augmentare a datelor și în ce ar trebui să constea aceasta. Iar în final voi descrie cum am ales dimensiunea rețelei finale ce va fi antrenată și testată cu hiperparametrii și politica de augmentare aleasă.

2.1. Modelul ResNet

O rețea reziduală (en. Residual Network – ResNet) este un model de arhitectură DL utilizat cel mai des în cadrul aplicațiilor de vedere artificială (en. Computer Vision). Este o arhitectură CNN proiectată să suporte sute sau chiar mii de straturi convoluționale. Arhitecturile precedente de CNN nu aveau capacitatea de a se adapta la numărul mare de straturi, lucru care conducea la o limitare a performanțelor pe măsura creșterii dimensiunii rețelei neurale. Prin adăugarea unui număr din ce în ce mai mare de straturi, cercetătorii s-au lovit de problema „risipirii gradienților” (en. vanishing gradients). [4]

Rețelele neurale sunt antrenate printr-un algoritm de backpropagation ce se bazează pe algoritmul de optimizare gradient descendent (en. gradient descent). Pe baza erorii găsite cu ajutorul funcției de cost, se procedează în ajustarea ponderilor în vederea minimizării erorii (în cazul lucrării de față, funcția de cost utilizată va fi entropia încrucișată – en. Cross Entropy Loss). Dacă numărul de straturi al rețelei este mult prea mare, înmulțirile multiple în procesul de propagare al erorii și de optimizare al ponderilor vor reduce eventual în mod considerabil gradienții, aceștia „risipindu-se”. În final, straturile aflate la începutul rețelei neurale vor primi ajustări mult prea mici, performanța totală a rețelei saturându-se sau chiar deteriorându-se cu fiecare strat adăugat suplimentar.

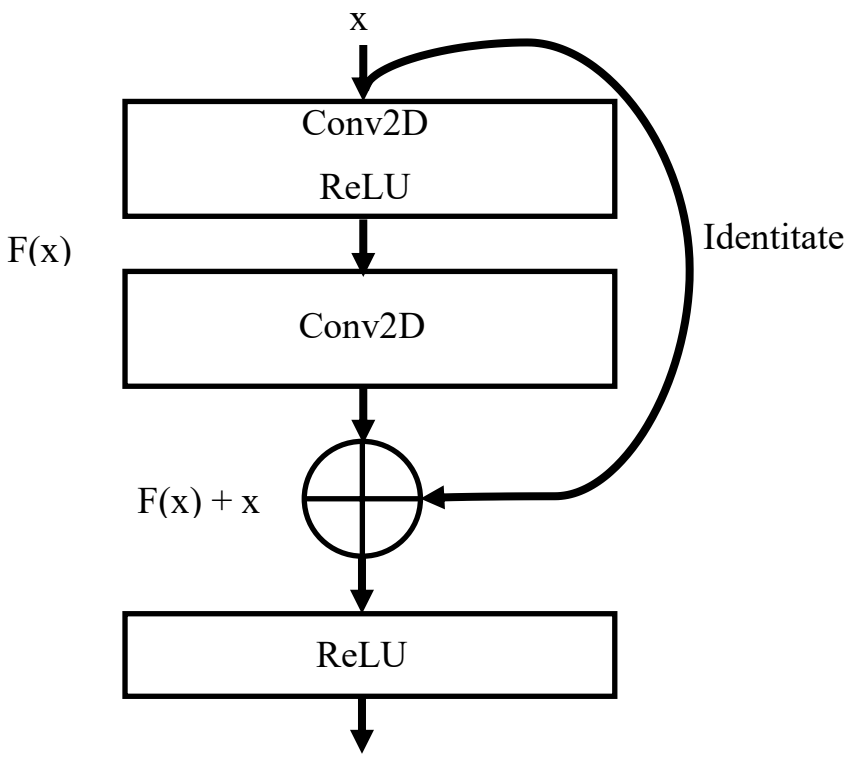
ResNet vine cu o soluție inovativă în vederea rezolvării problemei risipirii gradienților, cunoscută sub numele de „salturi de conexiune” (en. skip connections). ResNet suprapune mai multe mapări de identitate (salturi de conexiune), ce omit anumite blocuri reziduale formate din straturi convoluționale și reutilizează rezultatele blocului anterior, obținute în urma aplicării funcției de activare. Salturile accelerează procesul de antrenare inițial prin compresia rețelei neurale. Apoi, când rețeaua este reantrenată, toate straturile sunt utilizate, iar părțile rămase neantrenate din rețea, cunoscute sub numele de straturi reziduale, vor permite explorarea mult mai detaliată a spațiului caracteristicilor setului de date de intrare.

Blocurile Reziduale reprezintă inovația adusă de către arhitectura rețelelor ResNet. În arhitecturile anterioare precum VGG16 [5] sau LeNet [6] straturile convoluționale sunt suprapuse cu straturi de normalizare a lotului (en. batch normalization) și funcții de activare neliniare. Această metodă funcționează doar pentru arhitecturile cu număr mic de straturi convoluționale.

Arhitectura ResNet introduce un concept simplu, adăugarea de intrări intermediare la ieșirea unei serii de blocuri convoluționale. În cazul de față, în interiorul fiecărui bloc rezidual este realizată o conexiune cu ieșirea blocului rezidual anterior acestuia, fapt ce permite menținerea trăsăturilor (cunoștințelor) extrase anterior în cazul în care blocul respectiv nu aduce un plus de cunoștințe suficient de mare peste cel anterior. Astfel rețeaua permite

extragerea de trăsături și relații mult mai complexe în interiorul imaginilor optice satelitare, pe măsură ce rețeaua este antrenată în fiecare epocă.

Conexiunile dintre blocurile reziduale alăturate sunt realizate prin mapări de identitate directe (i.e. setul de trăsături rezultat la finalul blocului anterior este adunat la ieșirea blocului curent) în cazul blocurilor cu număr egal de filtre atât la intrare, cât și la ieșire, iar cazul blocurilor cu număr inegal de filtre, rețeaua prezintă la finalul fiecărui bloc un strat convoluțional ce realizează o ajustare a numărului de filtre pentru a permite adunarea acestora. Rețelele ResNet cu un număr de straturi mai mic de 50 se încadrează în prima categorie, în timp ce rețelele ResNet cu un număr de straturi mai mare de 50 se încadrează în categoria mapărilor de identitate ajustate cu un strat convoluțional.



Figură 2.1 Bloc Rezidual al ResNet cu salt peste acesta

Exact cum a fost menționat și anterior, rețelele neurale adânci precum VGG16 sau ResNet au straturile convoluționale suprapuse cu straturi de normalizare a loturilor și funcții de activare neliniare. Normalizarea loturilor este o tehnică de regularizare utilizată pentru a reduce varianța din interiorul datelor și a crește generalizarea soluției adusă de către rețea în tratarea problemei.

Tehnicile de regularizarea au rolul de a îmbunătății performanțele modelului și permit acestuia să conveargă mai rapid către soluția optimă. Câteva dintre instrumentele de regularizare sunt „early stopping”, „dropout”, tehnicile de inițializare a ponderilor și normalizarea loturilor. Regularizarea previne supraînvățarea modelului și face ca procesul de instruire să fie mai eficient. Despre supraînvățare se va discuta în paragraful 2.3.

Normalizarea este un instrument de preprocesare a datelor folosit pentru a aduce datele numerice la o scară comună, fără a le distorsiona forma. În general, atunci când sunt introduse date într-un algoritm de învățare profundă, apare tendința de a schimba valorile la o scară

echilibrată. Motivul pentru care datele sunt normalizate este pentru a asigura că modelul va putea generaliza în mod corespunzător.[7]

Revenind la normalizarea loturilor, acesta este un proces ce face rețelele neurale mai rapide și mai stabile prin adăugarea de straturi suplimentare într-o rețea neuronală adâncă. Noul strat, „BatchNorm", realizează operațiile de standardizare și normalizare a datelor de intrare provenite dintr-un strat anterior. Această operație are loc asupra unui întreg lot de date de intrare.

Un alt avantaj major adus de către normalizarea loturilor, pe lângă cele menționate anterior, este rezolvarea problemei „deplasării interne a covariatelor" (en. internal covariate shift). Prin această modalitate este asigurat că intrarea fiecărui strat este distribuită uniform în jurul aceleiași medii și abateri standard, rețeaua producând o generalizare mult mai bună.[7]

Capacitatea ridicată de învățare și generalizare a rețelei neurale de tip ResNet datorată arhitecturii specifice descrise anterior, produce o extragere extrem de eficientă a trăsăturilor din datele de intrare neprelucrate. Astfel, o rețea reziduală reprezintă soluția optimă pentru analiza și prelucrarea imaginilor ce conțin scene de teledetecție de o complexitate ridicată.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Tabel 1. Arhitecturi ResNet pentru setul de date ImageNet. Blocurile reziduale sunt descrise în parantezele pătrate ale tabelului alături de numărul de blocuri suprapuse. Subeșantionarea are loc în conv3_1, conv4_1, și conv5_1 cu un pas de 2.[8]

Lucrarea își propune antrenarea și testarea în fază incipientă a primelor 3 tipuri de ResNet, după mărimea lor: ResNet18, ResNet34 și ResNet50. Apoi, pe baza performanțelor obținute, se alege rețeaua ce oferă cea mai bună performanță în raport cu complexitatea acesteia (numărul de parametri).

2.2. Descrierea setului de date

Setul de date utilizat se numește UC Merced Land Use și conține imagini satelitare în spectrul RGB cu scene categorisite în 21 de clase, printre care: agricultural, airplane, baseballdiamond, beach, buildings, chaparral, denseresidential, forest, freeway, golfcourse, harbor, intersection și alte câteva astfel de scene frecvent întâlnite.

Fiecare clasă conține câte 100 de imagini drept exemplu, însumând 2100 de imagini în întregul set de date. Setul de date a fost compus în vederea măsurării performanței (benchmark) algoritmilor de Machine Learning, Deep Learning sau matematici în sarcina clasificării scenelor observate în imagini satelitare (teledetecție). Fiecare imagine are dimensiunea 256x256 pixeli.

Figură 2.2. Imagini exemplu din fiecare clasă - UC Merced Land Use

Împărțirea aleatoare a setului de date în date de antrenare, validare și testare este făcută în mod controlat, folosind un număr germene (en. seed) pentru a asigura reproductibilitatea experimentelor. De asemenea, datele sunt împărțite astfel încât pentru antrenare, validare și testare numărul de eșantioane din fiecare clasă să fie egal, evitând un eventual dezechilibru în timpul procesului de antrenare și validare a performanțelor.

2.3. Procesul de antrenare al rețelelor

Experimentele derulate utilizează o *dimensiune a lotului de date de intrare* (en. batch size) de *6 imagini optice* satelitare și un număr maxim de *80 de epoci de antrenare*.

Rata de învățare ce ajustează în urma fiecărei iterații prin setul de antrenare ponderile rețelei, pe baza erorii de clasificare, a fost *setată la* 10^{-3} știind că este recomandată alegerea unei rate de învățare cât mai mici, pentru a obține ajustări cât mai fine ale parametrilor în planul funcției de cost. Această valoare a fost aleasă suficient de mică încât să nu fie afectată de zgomotul funcției de cost (cauzat de calitatea setului de imagini), însă suficient de mare încât să aibă o convergență suficient de rapidă.

În procesul de învățare, pe lângă funcția de cost cross-entropy și optimizatorul Adam, se va mai folosi un programator al ratei de învățare (en. learning rate scheduler) ce va ajusta rata de învățare atunci când nu sunt detectate îmbunătățiri ale costului pe setul de validare. Programatorul ratei de învățare ReduceLROnPlateau (ro. reduce rata de învățare pe platou) va asigura că antrenarea nu se va opri prematur, în regiunile sub formă de platou ale funcției de cost, prin ajustarea ratei de învățare cu un factor de 0.1 atunci când performanța nu se îmbunătățește pentru un număr de epoci stabilit. Procesul de antrenare va continua, astfel, până la găsirea soluției celei mai apropiate de cea optimă global.

Metrica de performanță utilizată pentru determinarea costului rețelei neurale adânci în procesul de clasificare al scenelor de teledetecție este, în cazul lucrării de față, entropia încrucișată (en. Cross-Entropy loss). Entropia încrucișată este o măsură din domeniul teoriei informației, care se bazează pe entropie și, în general, calculează diferența dintre două distribuții de probabilitate. Este strâns legată, dar diferită de divergența Kullback–Leibler (KL) care calculează entropia relativă între două distribuții de probabilitate, în timp ce entropia încrucișată se poate traduce printr-un calcul al entropiei totale dintre distribuții. [9]

Antrenarea rețelei neurale adânci în condițiile unui timp limitat de rulare oferit în mod gratuit de către Google pentru sesiunile cu acces la resurse GPU va presupune salvarea stării la fiecare epocă de antrenare (en. checkpoint) pentru a putea relua ulterior procesul de antrenare când resursele vor redeveni disponibile. Salvarea stării procesului de antrenare presupune salvarea într-un dicționar a numărului epocii la care s-a realizat ultima salvare, salvarea dicționarului de stare al modelului, al optimizatorului, al programatorului ratei de învățare și salvare istoricului valorilor de cost în cadrul epocii respective.

Procesul de antrenare presupune în prima etapă preluarea unui lot de 6 imagini satelitare, trecerea lor pe dispozitivul utilizat (în cazul meu GPU) și calcularea estimărilor ieșirilor pentru fiecare imagine. În a doua etapă, pe baza estimărilor claselor rezultate și a valorilor observate (reale) se calculează funcția de cost, iar în funcție de valoarea obținută pentru entropia încrucișată se vor ajusta ponderile rețelei prin algoritmul de optimizare ales.

În bucla de antrenare (și validare) sunt cumulate costurile înregistrate pe parcursul epocii de antrenare pentru a putea calcula, la final, costul mediu înregistrat pe ambele seturi în cadrul epocii respective. Costurile medii ale epocii de antrenare servesc drept metrică de performanță a rețelei neurale și ajută la monitorizarea capacității rețelei de a rezolva problema dată și a găsi o soluție în spațiul datelor de intrare care să conducă la ieșirile dorite.

Pentru a evita situația în care rețeaua intră în supradaptare (en. overfitting) după un număr mare de epoci de antrenare, se introduce o metodă de regularizare (en. regularization) denumită „early stopping” (ro. oprire timpurie). Regularizarea este un proces ce ajustează soluțiile rezultate în cazul unor probleme de Machine Learning pentru a fi mai „simple” și a evita supradaptarea (sau supraînvățarea). Metoda de „early stopping” are rolul de a opri procesul de antrenare atunci când pentru un număr stabilit de epoci de antrenare nu se observă nicio îmbunătățire semnificativă a performanțelor sistemului.

Metoda de „early stopping” este implementată sub formă de clasă, ce conține un constructor cu parametrii, getteri și setteri pentru atributele ce se modifică în timp sau necesită preluarea lor pentru alte operații, metode de salvare și încărcare în Drive a valorilor costurilor din cadrul epoci cele mai performante și metoda de „earlyStop” ce are rolul de a verifica dacă performanțele rețelei s-au îmbunătățit sau nu în cadrul epocii în curs.

În cazul în care performanțele rețelei s-au îmbunătățit, atunci ponderile rețelei sunt salvate în Drive în format „.pt” prin instrucțiunea „torch.save”, costurile sunt salvate pentru

comparații ulterioare, numărul epocii este salvat, iar contorul ce reține numărul de epoci consecutive în care nu s-au detectat îmbunătățiri este resetat.

În cazul lucrării de față, numărul de epoci așteptate consecutiv până la oprirea procesului de învățare este 5. Acest număr a fost ales empiric, prin executarea mai multor experimente și observarea numărului maxim de epoci consecutive după care s-a detectat din nou o îmbunătățire semnificativă. Astfel se va economisi timp de rulare și nu se va antrena excesiv rețeaua neurală, obținând rezultatele optime pentru starea ponderilor rețelei.

Modalitatea de analiză a performanțelor pe setul de antrenare și cel de validare va consta în studierea evoluției entropiei încrucișate în cazul sarcinii de estimate a clasei, în funcție de numărul epocii de antrenare. În urma finalizării procesului de antrenare, pe baza istoricului valorilor entropiei încrucișate, se va trasa graficul valorilor funcției de cost în funcție de epoca de antrenare. Acest grafic va afișa care a fost progresul procesului de antrenare al rețelei neurale și cât de bine a fost îndeplinită sarcina de clasificare a scenelor urbane din vederi satelitare.

2.4. Augmentarea setului de date

Augmentarea datelor, prin transformări geometrice și fotometrice, este o practică standard de reducere a supraantrenării, ceea ce duce la o performanță mai bună de generalizare. Deoarece rețeaua noastră este concepută pentru a clasifica o scenă pe baza trăsăturilor extrase, nu toate transformările ar fi adecvate, deoarece caracteristicile din domeniul imaginii nu au întotdeauna interpretări fotometrice semnificative. Aplicarea unei reordonări a canalelor de culoare sau aplicarea unui anumit tip de zgomot pe scena satelitară captată nu vor ajuta la generalizarea problemei de clasificare a acelui tip de clasă. Tipul de zgomot și nivelul care ar trebui aplicat peste imagine depind de tipul aparatelor de achiziție ce vor fi utilizate cel mai adesea utilizate în această sarcină. Cum nu cunoaștem această informație, se va renunța la augmentarea datelor prin aplicarea unui zgomot sau prin inversarea canalelor de culoare.

Prin urmare, vom aplica cu o probabilitatea de 50% atât funcția geometrică de întoarcere a imaginii (flip), cât și cea de oglindire a acesteia. În urma experimentelor executate, s-a observat că rețeaua a reușit în final o generalizare mult mai bună a problemei identificării și clasificării scenelor terestre, datorită acestor transformări.

2.5. Procesul de testare al rețelelor neurale adânci

Procesul de testare al performanțelor rețelei neurale adânci se desfășoară în aceeași manieră precum antrenarea acesteia, însă printr-o singură iterație a întregului set de testare. La intrarea rețelei neurale sunt prezentate loturi de date de testare, sunt calculate predicțiile claselor, pe baza acestora sunt determinate entropiile încrucișate față de valorile observate (ideale), iar la final, suma tuturor erorilor este mediată la numărul de iterații efectuate prin set.

Etapa de testare a rețelei neurale vine cu noi metrici de performanță: precizia, reamintirea, scorul-f1 și acuratețea. Însă pentru a putea înțelege toate aceste metrici, trebuie luată în considere metrica de bază ce le definește pe acestea, matricea de confuzie.

O matrice de confuzie prezintă performanța unui model pe un set de date ținând cont de sarcina de clasificare pe care acesta trebuie să o îndeplinească. Pentru un set de date cu

scopul unei clasificări binară (care constă, să presupunem, din clase „pozitive” și „negative”), o matrice de confuzie are patru componente esențiale:

- **True pozitive** (TP): numărul de eșantioane prezise corect ca fiind „pozitive”;
- **False pozitive** (FP): numărul de eșantioane identificate în mod greșit ca fiind „pozitive”;
- **True Negative** (TN): numărul de eșantioane prezise corect ca fiind „negative”;
- **False Negative** (FN): numărul de eșantioane identificare în mod greșit ca fiind „negative”;

Acestea pot fi extrapolate și pentru cazul clasificării multiclasă, cum este cel de față, unde o clasificare TP este echivalentul unei clasificării în clasa corectă a imaginii aflate la intrarea rețelei, iar o clasificare TN este echivalentul unei clasificări corecte a unei imagini de la intrare în oricare altă clasă în afară de prima menționată (problema One-vs-All). De asemenea, o clasificare FP sau FN este opusul celor două menționate anterior.

Pe baza componentelor matricei de confuzie, în domeniul recunoașterii formelor, regăsirii informațiilor, detectării și clasificării obiectelor, se pot defini precizia și reamintirea. Acestea sunt metrici de performanță care se aplică datelor preluate dintr-un set de date.

Precizia (numită și valoare predictivă pozitivă) este fracția de instanțe relevante dintre instanțele recuperate. Scris ca o formulă:

$$Precision = \frac{TP}{TP + FP}$$

Reamintirea (cunoscută și ca sensibilitate) este fracțiunea de instanțe relevante care au fost preluate. Scris sub formă de formulă:

$$Recall = \frac{TP}{TP + FN}$$

Scorul F1 este calculat ca medie armonică a scorurilor de precizie și de reamintire, după cum se arată mai jos. Acesta variază de la 0 la 100%, iar un scor F1 mai mare denotă un clasificator de calitate mai bună.

$$F1\ Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

Scorul F1 macromediat al unui model este doar o medie simplă a scorurilor F1 la nivel de clasă obținute. Scorul F1 macro-mediat este util numai atunci când setul de date utilizat are același număr de eșantioane de date în fiecare dintre clasele sale. Cu toate acestea, cele mai multe seturi de date din lumea reală sunt dezechilibrate de clasă - categorii diferite au cantități diferite de date. În astfel de cazuri, o medie simplă poate fi o valoare de performanță înșelătoare. În cazul lucrării de față, avem un număr egal de eșantioane pentru fiecare clasă, prin urmare voi aplica macromedierea rezultatelor tuturor metricilor.

Scorul F1 micro-mediat este o valoare care are sens pentru distribuțiile de date cu mai multe clase. Utilizează valori „net” TP, FP și FN pentru calcularea valorii. TP net se referă la suma scorurilor TP la nivel de clasă ale unui set de date, care sunt calculate prin dizolvarea unei matrice de confuzie în matrice one-vs-all corespunzătoare fiecărei clase. Pentru un set de date de clasă binară, un scor micro F1 este pur și simplu scorul de precizie

3. Desfășurarea experimentelor și rezultate experimentale

Toate experimentele realizate în această lucrare beneficiază de o inițializare a parametrilor modelelor cu valori preantrenate pe setul de date ImageNet, tehnică de regularizare denumită transfer learning (ro. transferul învățării). ImageNet este un set de date gigant ce este destinat clasificării imaginilor în 20000 de clase, fiecare clasă conținând 1000 de eșantioane. Cunoștințele extrase prin antrenarea unei rețele neurale adânci pe acest set de date sunt de înțelegere a obiectului/subiectului prezentat în scenă și de extragere a trăsăturilor definitorii ale acestuia (i.e. contururi, texturi, culori etc.). Prin aplicarea metodei de transfer a cunoștințelor, procesul de antrenare este accelerat și este asigurată o convergență a valorilor ponderilor către valoarea optimă ce va asigura clasificarea imaginilor satelitare în tipul de scenă adnotat. Acest lucru este garantat de către faptul că atât clasificarea subiecților din ImageNet, cât și a scenelor din UC Merced Land Use sunt sarcini foarte similare.

3.1. Antrenare cu sau fără augmentarea setului date

Primul experiment a constat în determinarea eficienței metodelor de augmentare, comparativ cu o metodă clasică de antrenare a rețelei neurale, direct pe setul de antrenare nealterat. Antrenarea s-a desfășurat fără utilizarea unui early stopper sau a unui set de validare, îndeplinind toate cele 80 de epoci.

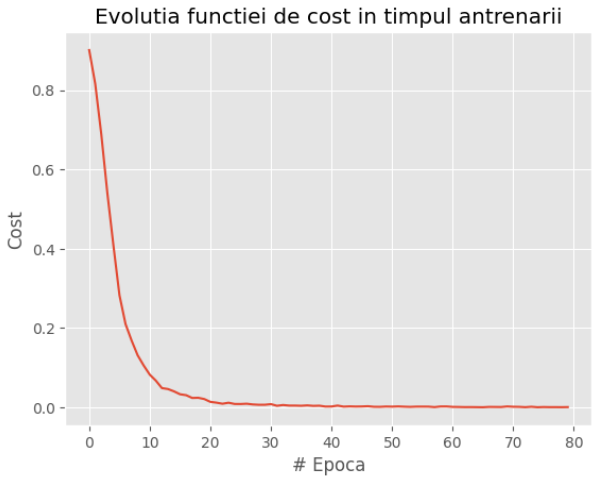
Rezultatele obținute atât pentru setul de date augmentat cât și neaugmentat sunt pentru modelul ResNet50. Acest experiment va urmări pe lângă sarcina menționată anterior, și determinarea numărului optim de epoci de antrenare pentru acest model de rețea neurală adâncă. Întregul set de date de 2100 de imagini a fost împărțit în 20% date de testare, iar restul pentru antrenare.

În continuare, sunt prezentate și comentate rezultatele procesului de antrenare pe setul de date neaugmentat:

Număr epoci	Cost antrenare	Cost testare	Precizie/Reamintire/Acuratețe
20	0.02098	0.2118	0.97 / 0.96 / 0.96
40	0.00204	0.2867	0.95 / 0.94 / 0.94
60	0.00232	0.2642	0.96 / 0.95 / 0.95
80	0.00071	0.1851	0.97 / 0.96 / 0.96

Tabel 2. Rezultatele antrenării rețelei pe un set neaugmentat

Rezultatele obținute relevă faptul că modelul ResNet50 învață doar în primele 20 de epoci, după aceea procesul fiind unul divergent, obținând o generalizare mult mai slabă. În această epocă se obține o valoare de cost suficient de mică pe setul de testare și pe cel de antrenare, precum și cele mai bune valori pentru metricile de performanță precizie, reamintire și acuratețe. Se mai poate distinge o mică îmbunătățire a valorii funcției de cost pe setul de testare în epoca 80, însă nu este semnificativă în comparație cu cea din epoca 20.



Figură 3.1. Evoluția funcției de cost pentru setul neaugmentat

De asemenea, ar trebui avută în vedere și diferența drastică dintre valoarea funcției de cost pe setul de antrenare față de cel de testare, aceasta fiind mai mică chiar cu trei ordine de mărime în cadrul ultimei epoci. Acest lucru semnalează faptul că modelul ResNet50 are o capacitate foarte bună de memorare a soluției problemei datorită erorii mici pe setul de antrenare, însă generalizarea pe setul de validare este mult mai slabă. Prin urmare, cea mai bună generalizare se obține chiar din primele 20 de epoci, diferența dintre valoarea funcției de cost pe setul de antrenare și testare fiind doar de un ordin de mărime. După această epocă, procesul de generalizare al soluției pe un set de date necunoscut se oprește, fapt ce se reflectă atât în valorile mai crescute ale funcției de cost, cât și în valorile mai slabe pentru precizie, reamintire și acuratețe.

Aceste lucruri sunt confirmate și de către graficul curbei funcției de cost pe setul de antrenare în funcție de numărul epocii de antrenare. După epoca 20, scăderea valorii funcției de cost nu mai este la fel de semnificativă, evidențiind faptul că rețeaua nu mai generalizează la fel de bine, trecând la un proces de memorare al soluțiilor setului de antrenare.

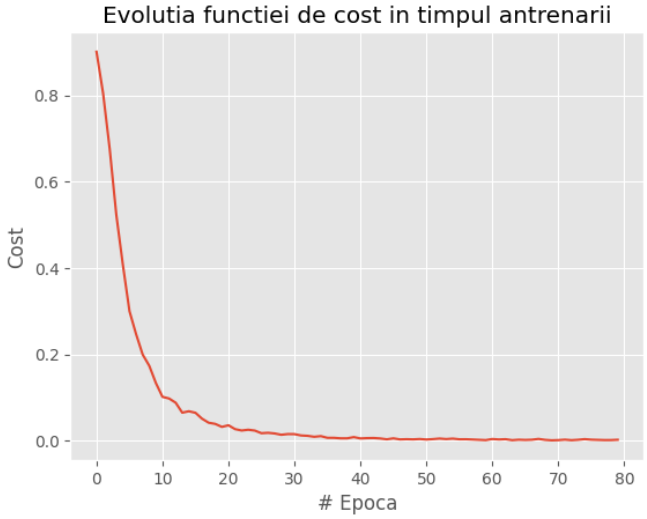
În continuare vor fi prezentate și comentate rezultatele procesului de antrenare al modelului folosind metodele de augmentare menționate în paragraful 2.4.

Număr epoci	Cost antrenare	Cost testare	Precizie/Reamintire/Acuratețe
20	0.03203	5.5441	0.93 / 0.91 / 0.91
40	0.00843	2.4701	0.96 / 0.95 / 0.95
60	0.00140	0.7583	0.96 / 0.95 / 0.95
80	0.00223	2.4563	0.96 / 0.96 / 0.96

Tabel 3. Rezultatele antrenării pentru un set de date augmentat

Rezultatele arată că metodele de augmentare utilizate, ambele cu o probabilitate de 50% de a fi aplicate, impun o nouă provocare în procesul de antrenare al ponderilor rețelei. Efectul se reflectă în valorile funcției de cost mai mari pe setul de antrenare, comparativ cu cazul anterior, dar și în valorile funcției de cost pe setul de testare. Chiar și după 60 de epoci, rețeaua nu reușește să generalizeze la fel de bine ca în cazul setului de date neaugmentat (ce are o eroare pe setul de testare mult mai mică). Cu toate acestea, valorile pentru precizie, reamintire și acuratețe sunt mult mai bune și mult mai echilibrate față de cazul setului de antrenare neîmbogățit.

Graficul evoluției funcției de cost în timpul procesului de antrenare arată că rețeaua își continuă pentru mai mult timp învățarea, începând abia de la epoca 40 să nu mai obțină îmbunătățiri semnificative în cunoștințe pe setul de antrenare. De asemenea, generalizarea se oprește tot aici, valorile pentru precizie, reamintire și acuratețe pe setul de test schimbându-se doar cu un procent până la epoca 80.



Figură 3.2. Evoluția funcției de cost pentru setul augmentat

Prin urmare, având în vedere rezultatele prezentate pentru setul de date augmentat și observațiile formei graficului ce indică un proces de învățare extins față de utilizarea setului de date neaugmentat (acesta oprindu-se la epoca 20), se va continua studiul folosindu-mă de plusul de cunoștințe adus de către metodele de augmentare propuse. Astfel, se vor obține rezultate cu o generalizare mult mai largă pentru setul de date neutilizat în timpul procesului de antrenare.

3.2. Antrenare rețelei ResNet în vederea deciziei modelului

În primul rând, în urma rezultatelor experimentale obținute în paragraful anterior ce prezintă o inconsistență în numărul optim de epoci de antrenare, pentru a putea găsi modelul convoluțional optim pentru această sarcină am ales utilizarea early stopper-ului și a unui procent din setul de antrenare ca set de validare al performanțelor obținute în timpul procesului de antrenare. Aceste lucruri, împreună cu politica de augmentare aleasă, vor asigura cea mai bună generalizare alături de o antrenare controlată a modelelor testate. În plus, datorită utilizării setului de validare ce permite monitorizarea performanței rețelei pe un set de date necunoscut rețelei convoluționale, se va putea ajusta și rata de învățare în timpul antrenării cu ajutorul unui programator al ratei de învățare precum cel prezentat în paragraful 2.3.

Prin urmare, setul de date a fost împărțit în 15% date de testare, 15% date de validare și restul datelor de antrenare. Folosind această împărțire a setului de date alături de noile tehnici de regularizare, au fost antrenate și testate modelele ResNet18, 34 și 50. Rezultatele sunt prezentate în tabelul următor.

Programatorul ratei de învățare a fost setat ca după 3 epoci de stagnare a valorii funcției de cost pe setul de antrenare să scadă rata de învățare cu un ordin de mărime, iar early stopper-ul va opri antrenarea după 5 epoci de stagnare a valorii funcției de cost pe setul de validare. Epoca finală din tabelele următoare se referă la epoca la care a avut lot ultima îmbunătățire a valorii funcției de cost pe setul de validare. Pentru determinarea duratei totale de antrenare, se adaugă cele 5 epoci de așteptare setate pentru oprirea timpurie a antrenării.

Tipul modelului	Epoca Finală	Cost Antrenare	Cost Validare
ResNet18	23	0.01737	0.08932
ResNet34	25	0.01976	0.10946
ResNet50	12	0.04152	0.11289

Tabel 4. Rezultatele antrenării pentru modelele ResNet 18, 34 și 50

Tipul modelului	Cost set testare	Precizie	Reamintire	Acuratețe
ResNet18	0.0635	0.99	0.98	0.98
ResNet34	0.1030	0.97	0.97	0.97
ResNet50	0.1919	0.97	0.97	0.97

Tabel 5. Rezultatele testării pentru modelele ResNet 18, 34, 50

Rezultatele au arătat că nu este necesară utilizarea unui model de dimensiuni medii sau mari, precum este ResNet34 sau 50. Valorile cele mai bune obținute atât timpul procesului de antrenare, cât și cel de testare, au fost cu ajutorul celui mai mic model, ResNet18. Acest lucru este cel mai probabil datorat atât de către dimensiunea redusă a setului de antrenare, cât și de către simplitatea sarcinii de clasificare a scenelor extrase din imagini de teledetecție (rezumându-se în mare parte la clasificarea de texturi prezente în imagine și înțelegerea scenei prin identificare obiectelor predominant prezente în ea și poziționarea lor).

Această teorie a fost testată și confirmat prin aplicarea unor descriptori simpli asupra acestui set de date, Histograma Alb-Negru și Histograma Gradienților Orientați (Histogram of Oriented Gradients - HoG), alături de doi clasificatori clasici, k-NN și Random Forest. Rezultatele au arătat că cea mai bună clasificare a fost realizată cu ajutorul a 500 de arbori de clasificare și descriptorul HoG, obținând ~63% scor de clasificare pe setul de testare.

Nu în ultimul rând, acest test a confirmat faptul că utilizarea politicii de augmentare propuse alături de metodele de regularizare vor duce la rezultate sporite. Acest lucru se poate observa în îmbunătățirea valorilor funcției de cost pe toate cele trei seturile alături de îmbunătățirea metricilor de performanță precizie, reamintire și acuratețe cu un procent pentru ResNet.

3.3. Antrenare rețelei ResNet18 în vederea validării rezultatelor obținute

Continuând cu modelul care a obținut cea mai bună performanță în paragraful anterior, ResNet18, voi efectua câteva experimente suplimentare pentru a valida a performanța obținută anterior prin schimbarea proporției de date de testare și validare în raport cu restul datelor

rămase pentru antrenare. Primul experiment a constat în prelevarea a 20% eșantioane pentru testare și 20% pentru validare, iar pentru ultimul experiment am împărțit 10% din date pentru validare și 20% pentru testare. Rezultatele obținute pentru ambele scenarii sunt prezentate în tabelul următor.

Tipul de împărțire Validare / Testare	Epoca finală	Cost Antrenare	Cost Validare
20% / 20%	11	0.0551	0.1237
10% / 20%	17	0.0275	0.0642
15% / 15%	23	0.01737	0.08932

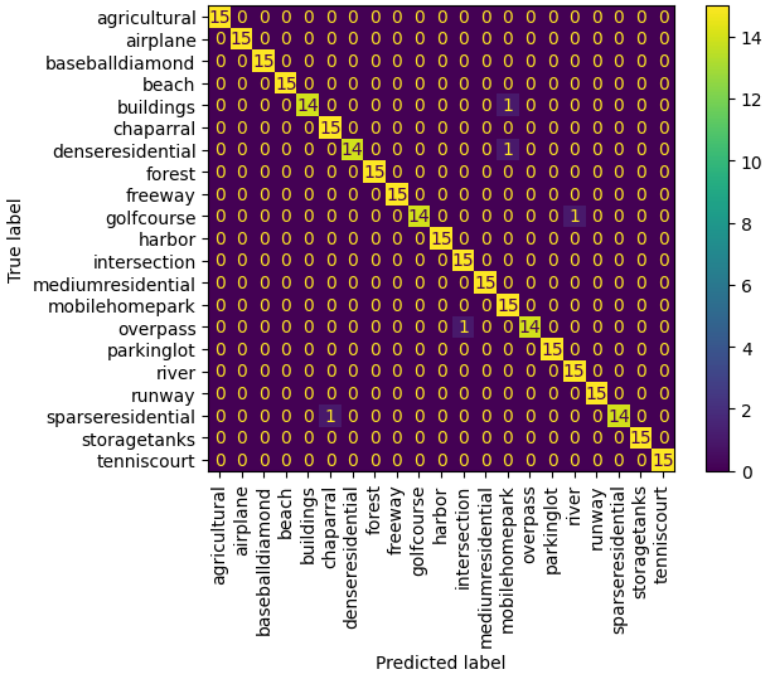
Tabel 6. Rezultatele antrenării pentru împărțiri diferite ale setului de date

Tipul de împărțire Validare / Testare	Cost Testare	Precizie	Reamintire	Acuratețe
20% / 20%	0.1071	0.97	0.97	0.97
10% / 20%	0.0626	0.98	0.98	0.98
15% / 15%	0.0635	0.99	0.98	0.98

Tabel 7. Rezultatele testării pentru împărțiri diferite ale setului de date

Cele două experimente au demonstrat că atribuirea unui procent mai mare din datele de antrenare către validare sau testare nu afectează semnificativ rezultatul antrenării sau performanța modelului pe setul de testare, scăzând cu maxim 2 procente metricile de performanță sau cu câteva unități valoarea finală a funcției de cost.

În final, voi prezenta pentru cea mai bună împărțire a datelor, adică împărțirea inițială 15% validare, 15% testare, și pentru cel mai bun model, ResNet18, matricea de confuzie pe setul de date de testare, set ce are împărțit în mod egal eșantioanele între cele 21 de clase.



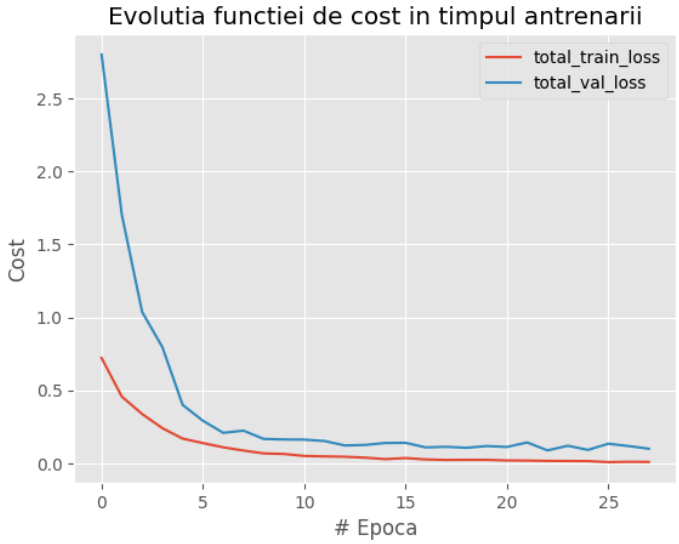
Figură 3.3. Matricea de confuzie pentru ResNet18

Din matricea de confuzie, reiese faptul că în setul de testare au fost atribuite 15 imagini pentru fiecare clasă, rezultând în total 315 imagini pentru întregul set de date de test. Pentru performanțele obținute anterior, se poate observa că rețeaua confundă doar 5 imagini (clasificare eronată). Clasele care sunt confundate de către rețea, sunt clase ce au trăsături extrem de similare, fiind de așteptat să aibă loc un astfel de fenomen: de exemplu, buildings (ro. clădiri) sau denseresidentsial (ro. zonă rezidențială densă) sunt confundate cu clasa mobilehomepark (ro. parc de autorulote).

	precision	recall	f1-score	support
agricultural	1.00	1.00	1.00	15
airplane	1.00	1.00	1.00	15
baseballdiamond	1.00	1.00	1.00	15
beach	1.00	1.00	1.00	15
buildings	1.00	0.93	0.97	15
chaparral	0.94	1.00	0.97	15
denseresidential	1.00	0.93	0.97	15
forest	1.00	1.00	1.00	15
freeway	1.00	1.00	1.00	15
golfcourse	1.00	0.93	0.97	15
harbor	1.00	1.00	1.00	15
intersection	0.94	1.00	0.97	15
mediumresidential	1.00	1.00	1.00	15
mobilehomepark	0.88	1.00	0.94	15
overpass	1.00	0.93	0.97	15
parkinglot	1.00	1.00	1.00	15
river	0.94	1.00	0.97	15
runway	1.00	1.00	1.00	15
sparseresidential	1.00	0.93	0.97	15
storagetanks	1.00	1.00	1.00	15
tenniscourt	1.00	1.00	1.00	15
accuracy			0.98	315
macro avg	0.99	0.98	0.98	315
weighted avg	0.99	0.98	0.98	315

Tabel 8. Prezentare detaliată pe clase a metricilor de performanță - ResNet18

În ciuda modelului de dimensiuni mici ce a fost utilizat, clasificarea realizată de către acesta se ridică la nivelul performanței rețelei state-of-the-art, ResNet50, antrenate pentru clasificarea imaginilor satelitare din acest set de date cu o acuratețe de 99.6%. Modelul ResNet18 (18 straturi), o versiune cu un număr mult mai redus de parametri față de ResNet50 (50 de straturi) reușește datorită metodelor de augmentare propuse și a metodelor de regularizare utilizate să egaleze într-o oarecare măsură performanța rețelei state-of-the-art.



Figură 3.4. Evoluția funcției de cost pentru antrenarea ResNet18

Trebuie menționat și timpul de antrenare foarte scurt al modelului ResNet18. Întregul proces a durat doar 997 de secunde sau aproximativ 15 minute, față de ResNet50 care a durat 1079 de secunde pentru un număr de epoci mult mai mic, doar 17 epoci în comparație cu 28 pentru ResNet18. Acest lucru s-a datorat atât metodelor de regularizare utilizate, cât și parametrilor preantrenați pe setul de date ImageNet (tehnica de transfer a cunoștințelor) pentru un model cu timp de inferență redusă precum ResNet18. Experimentele desfășurate pe platforma Google Colab au beneficiat de accelerare GPU, folosind o placă grafică NVIDIA Tesla T4.

4. Concluzii

Întregul experiment a demonstrat, în esență, că nu întotdeauna este necesar un model cu învățare profundă foarte complex pentru obținerea unor rezultate bune pentru rezolvarea sarcinilor dorite. În cazul de față, pentru un set de date de dimensiuni mici, ce conține imagini de teledetecție de dimensiuni medii, o rețea precum ResNet18 este suficient de bună pentru a obține rezultate ce pot egala metodele state-of-the-art existente. Utilizarea unor metode de augmentare a setului de date, ce conduc la o mărire artificială a numărului de eșantioane pentru antrenare și testare, ajută la generalizarea mult mai bună a problemei și obținerea unor rezultate finale mult mai relevante. În ultimul rând, metodele de regularizare utilizate și transferul cunoștințelor prin inițializarea rețelei cu ponderi preantrenate fac ca procesul de antrenare să aibă o convergență mult mai stabilă, mai rapidă și mai sigură către minimul global al funcției de cost utilizate.

Bibliografie

[1] V.-E. Neagoe, “INTRODUCERE IN DEEP LEARNING Modelul Deep Learning (DL),” 2023.

[2] “7.2. Convolutions for Images — Dive into Deep Learning 1.0.0-beta0 documentation.” Accessed: Jun. 21, 2023. [Online]. Available: https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html

[3] “Max Pooling Explained | Papers With Code.” Accessed: Jun. 19, 2023. [Online]. Available: https://paperswithcode.com/method/max-pooling

[4] “Various Optimization Algorithms For Training Neural Network | by Sanket Doshi | Towards Data Science.” Accessed: Jun. 10, 2023. [Online]. Available: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

[5] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Sep. 2014, Accessed: Jun. 10, 2023. [Online]. Available: https://arxiv.org/abs/1409.1556v6

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[7] “Batch Normalization | What is Batch Normalization in Deep Learning.” Accessed: Jun. 20, 2023. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/

[8] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition”, Accessed: May 08, 2024. [Online]. Available: http://image-net.org/challenges/LSVRC/2015/

[9] “A Gentle Introduction to Cross-Entropy for Machine Learning - MachineLearningMastery.com.” Accessed: May 08, 2024. [Online]. Available: https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Anexă

1. Importul bibliotecilor și conectarea la Google Drive:

```
import os.path
import torch
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import transforms
import torch.nn.functional as F
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
from torch.nn import (Module, Conv2d, Linear, BatchNorm2d, Sequential, ReLU, MaxPool2d,
                      AdaptiveAvgPool2d, Dropout, AvgPool2d, CrossEntropyLoss)
import torch.nn.functional as F
import numpy as np
import torch
import time
import random
from torch.optim.lr_scheduler import ReduceLROnPlateau

from google.colab import drive
drive.mount('/content/drive')
%cd drive/My Drive/UCMerced_LandUse
models_dir = './model_dir'
```

2. Politica de augmentare a setului de date:

```
class BasicPolicy(object):
  def __init__(self, mirror_ratio=0, flip_ratio=0, color_change_ratio=0,
is_full_set_colors=False, add_noise_peak=0.0, erase_ratio=-1.0):
    # Random color channel order
    from itertools import product, permutations
    self.indices = list(product([0, 1, 2], repeat = 3)) if is_full_set_colors else
list(permutations(range(3), 3))
    self.indices.insert(0, [0, 1, 2])# R, G, B
    self.add_noise_peak = add_noise_peak

    # Mirror and flip
    self.color_change_ratio = color_change_ratio
    self.mirror_ratio = mirror_ratio
    self.flip_ratio = flip_ratio
```

```
    # erase
    self.erase_ratio = erase_ratio

  def __call__(self, img):
    # 0) Add poisson noise (e.g. choose peak value 20)
    # https://stackoverflow.com/questions/19289470/adding-poisson-noise-to-an-image
    if self.add_noise_peak > 0:
          PEAK = self.add_noise_peak
          img = np.random.poisson(np.clip(img, 0, 1) * PEAK) / PEAK

    # Color change
    policy_idx = random.randint(0,len(self.indices)-1)
    if random.uniform(0, 1) >= self.color_change_ratio:
      policy_idx = 0

    img = img[...,list(self.indices[policy_idx])]

    # Mirror image
    if random.uniform(0,1) <= self.mirror_ratio:
      img = img[...,::-1,:].copy()

    # Flip image vertically
    if random.uniform(0, 1) < self.flip_ratio:
      img = img[::-1,:,:].copy()

    # 4) Erase random box
    if random.uniform(0, 1) < self.erase_ratio:
      img = self.eraser(img)

    return img

  def __repr__(self):
    return "Basic Policy"

  def eraser(self, input_img, p=0.5, s_l=0.02, s_h=0.4, r_1=0.3, r_2=1/0.3, v_l=0, v_h=255,
pixel_level=True):
    img_h, img_w, img_c = input_img.shape
    p_1 = np.random.rand()

    if p_1 > p:
      return input_img

    while True:
      s = np.random.uniform(s_l, s_h) * img_h * img_w
      r = np.random.uniform(r_1, r_2)
      w = int(np.sqrt(s / r))
      h = int(np.sqrt(s * r))
      left = np.random.randint(0, img_w)
```

```
      top = np.random.randint(0, img_h)

      if left + w <= img_w and top + h <= img_h:
        break

    if pixel_level:
        c = np.random.uniform(v_l, v_h, (h, w, img_c))
    else:
        c = np.random.uniform(v_l, v_h)

    input_img[top:top + h, left:left + w, :] = c

    return input_img
```

3. Dataloader pentru setul UC Merced Land Use:

```
class UCMercedLandUseDataset(Dataset):
    def __init__(self, root_dir, transform=None, is_flip=False, is_addnoise=False,
is_erase=False):
        """
        Args:
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied on a sample.
        """
        self.root_dir = root_dir
        self.transform = transform

        self.images = []
        self.labels = []
        self.policy = BasicPolicy(color_change_ratio=0, mirror_ratio=0.50, flip_ratio=0.5 if
not is_flip else 0.2,
                                  add_noise_peak=0 if not is_addnoise else 20, erase_ratio=-
1.0 if not is_erase else 0.5)

        # Load dataset
        self.load_dataset()

    def load_dataset(self):
       # assuming classes are folders in the root_dir
       self.classes = sorted(os.listdir(self.root_dir))

       for class_index, class_name in enumerate(self.classes):
        class_dir = os.path.join(self.root_dir, class_name)
        if not os.path.isdir(class_dir):
          continue

        for image_name in os.listdir(class_dir):
```

```
            if image_name.lower().endswith(('png', 'jpg', 'jpeg', 'tif')):
              image_path = os.path.join(class_dir, image_name)
              self.images.append(image_path)
              self.labels.append(class_index)

    def __len__(self):
      return len(self.images)

    def __getitem__(self, index, is_apply_policy=True):
      image_path = self.images[index]
      image =  np.clip(np.asarray(Image.open(image_path).convert('RGB'))/255,0,1)
      label = self.labels[index]

      if is_apply_policy: image = self.policy(image)

      if self.transform:
        image = self.transform(image)

      return image.float(), label
```

4. Împărțirea setului de date în antrenare, validare și testare:

```
# ensuring even split of classes in train and test sets
def stratified_split(dataset, test_size=0.2, validation_size=0.1, random_state=None):
  # Assuming 'dataset.labels' contains your labels
  labels = dataset.labels

  # Create stratified split for test set
  train_index, test_index = train_test_split(
      range(len(labels)),
      test_size=test_size,
      random_state=random_state,
      stratify=labels)

  # Adjust validation size based on remaining data after test split
  adjusted_validation_size = validation_size / (1 - test_size)

  # Create stratified split for validation and train sets
  train_index, val_index = train_test_split(
      train_index,
      test_size=adjusted_validation_size,
      random_state=random_state,
      stratify=[labels[i] for i in train_index])

  return train_index, val_index, test_index
```

5. Definirea modelului ResNet:

```
class Bottleneck (Module):
    def __init__(self, in_channels, intermediate_channels, expansion, is_Bottleneck, stride):
        """
        Creates a Bottleneck with conv 1x1->3x3->1x1 layers.

        Note:
            1. Addition of feature maps occur at just before the final ReLU with the input
feature maps
            2. if input size is different from output, select projected mapping or else
identity mapping.
            3. if is_Bottleneck=False (3x3->3x3) are used else (1x1->3x3->1x1). Bottleneck is
required for resnet-50/101/152
        Args:
            in_channels (int) : input channels to the Bottleneck
            intermediate_channels (int) : number of channels to 3x3 conv
            expansion (int) : factor by which the input #channels are increased
            stride (int) : stride applied in the 3x3 conv. 2 for first Bottleneck of the block
and 1 for remaining

        Attributes:
            Layer consisting of conv->batchnorm->relu
        """

        super(Bottleneck, self).__init__()

        self.expansion = expansion
        self.in_channels = in_channels
        self.intermediate_channels = intermediate_channels
        self.is_Bottleneck = is_Bottleneck

        # i.e. if dim(x) == dim(F) => identity function
        if self.in_channels == self.intermediate_channels * self.expansion:
            self.identity = True

        else:
            self.identity = False
            downsample_layer = [
                Conv2d(in_channels=self.in_channels, out_channels=self.intermediate_channels *
self.expansion,
                       kernel_size=1, stride=stride, padding=0, bias=False),
                BatchNorm2d(self.intermediate_channels * self.expansion)]
            # Only conv -> BN and no ReLU
            # projection layer.append(ReLU())
            self.downsample= Sequential(*downsample_layer)

        self.relu = ReLU()
```

```
        # is_Bottleneck = True for all ResNet50+
        if self.is_Bottleneck:
            # bottleneck
            # 1x1
            self.conv1 = Conv2d(in_channels=self.in_channels,
out_channels=self.intermediate_channels, kernel_size=1,
                                    stride=1, padding=0, bias=False)
            self.bn1 = BatchNorm2d(self.intermediate_channels)

            # 3x3
            self.conv2 = Conv2d(in_channels=self.intermediate_channels,
out_channels=self.intermediate_channels,
                                    kernel_size=3, stride=stride, padding=1, bias=False)
            self.bn2 = BatchNorm2d(self.intermediate_channels)

            # 1x1
            self.conv3 = Conv2d(in_channels=self.intermediate_channels,
out_channels=self.intermediate_channels*self.expansion,
                                    kernel_size=1, stride=1, padding=0, bias=False)
            self.bn3 = BatchNorm2d(self.intermediate_channels*self.expansion)

        else:
            # basicblock
            # 3x3
            self.conv1 = Conv2d(in_channels=self.in_channels,
out_channels=self.intermediate_channels, kernel_size=3,
                                    stride=stride, padding=1, bias=False)
            self.bn1 = BatchNorm2d(self.intermediate_channels)

            # 3x3
            self.conv2 = Conv2d(in_channels=self.intermediate_channels,
out_channels=self.intermediate_channels,
                                    kernel_size=3, stride=1, padding=1, bias=False)
            self.bn2 = BatchNorm2d(self.intermediate_channels)

    def forward(self, x):
        # input stored to be added before final ReLU - skip connection
        in_x = x

        if self.is_Bottleneck:
            # conv1x1->BN->ReLU
            x = self.relu(self.bn1(self.conv1(x)))

            # conv3x3->BN->ReLU
            x = self.relu(self.bn2(self.conv2(x)))

            # conv1x1->BN
            x = self.bn3(self.conv3(x))
```

```
        else:
            # conv3x3->BN->ReLU
            x = self.relu(self.bn1(self.conv1(x)))

            #conv3x3->BN
            x = self.bn2(self.conv2(x))

        if self.identity:
            x += in_x
        else:
            x += self.downsample(in_x)

        # final ReLU
        x = self.relu(x)

        return x


class ResNet(Module):
    def __init__(self, resnet_variant, in_channels, num_classes=None,
feature_extractor=False):
        """
        Creates the ResNet architecture based on the provided variant. 18/34/50/101 etc.
        Based on the input parameters, define the channels list, repetition list along with
expansion factor(4) and stride(3/1)
        using _make_blocks method, create a sequence of multiple Bottlenecks
        Average Pool at the end before the FC layer

        Args:
            resnet_variant (list) : e.g. [[64,128,256,512],[3,4,6,3],4,True]
            in_channels (int) : image channels (3)
            num_classes (int) : output #classes

        Attributes:
            Layer consisting of conv->batchnorm->relu

        """

        super(ResNet, self).__init__()
        print(f"[INFO] initializing the ResNet model...")
        self.channel_list = resnet_variant[0]
        self.repetition_list = resnet_variant[1]
        self.expansion = resnet_variant[2]
        self.is_Bottleneck = resnet_variant[3]
        self.feature_extractor = feature_extractor

        if not self.feature_extractor and num_classes is None:
            raise ValueError(f"num_classes must be of type integer, got None instead.")
```

```
        self.conv1 = Conv2d(in_channels=in_channels, out_channels=64, kernel_size=7, stride=2,
padding=3, bias=False)
        self.bn1 = BatchNorm2d(64)
        self.relu = ReLU()

        self.maxpool = MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_blocks(64, self.channel_list[0], self.repetition_list[0],
self.expansion,
                                        self.is_Bottleneck, stride=1)
        self.layer2 = self._make_blocks(self.channel_list[0]*self.expansion,
self.channel_list[1], self.repetition_list[1],
                                        self.expansion, self.is_Bottleneck, stride=2)
        self.layer3 = self._make_blocks(self.channel_list[1]*self.expansion,
self.channel_list[2], self.repetition_list[2],
                                        self.expansion, self.is_Bottleneck, stride=2)
        self.layer4 = self._make_blocks(self.channel_list[2]*self.expansion,
self.channel_list[3], self.repetition_list[3],
                                        self.expansion, self.is_Bottleneck, stride=2)

        if not self.feature_extractor:
            self.average_pool = AdaptiveAvgPool2d(1)
            self.fc1 = Linear(self.channel_list[3]*self.expansion, num_classes)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        if not self.feature_extractor:
            x = self.average_pool(x)
            x = torch.flatten(x, start_dim=1)
            x = self.fc1(x)

        return x

    def _make_blocks(self, in_channels, intermediate_channels, num_repeat, expansion,
is_Bottleneck, stride):
        """
        Args:
            in_channels : #channels of the Bottleneck input
            intermediate_channels : #channels of the 3x3 in the Bottleneck
            num_repeat : #Bottlenecks in the block
```

```
            expansion : factor by which intermediate_channels are multiplied to create the
output channels
            is_Bottleneck : status if Bottleneck in required
            stride : stride to be used in the first Bottleneck conv 3x3

        Attributes:
            Sequence of Bottleneck layers

        """

        layers = [Bottleneck(in_channels, intermediate_channels, expansion, is_Bottleneck,
stride=stride)]

        for num in range(1, num_repeat):
            layers.append(Bottleneck(intermediate_channels*expansion, intermediate_channels,
expansion, is_Bottleneck,
                                     stride=1))

        return Sequential(*layers)
```

6. Definirea funcției de Early Stop:

```
# defining Early Stopping class
class EarlyStopping():
  def __init__(self, patience=1, min_delta=0):
    self.patience = patience
    self.min_delta = min_delta
    self.counter = 0
    self.min_validation_loss = np.inf
    self.best_epoch = 0
    self.best_train = None
    self.best_val = None

  def earlyStop(self, epoch, trainLoss, valLoss, model):
    if valLoss <= (self.min_validation_loss + self.min_delta):
      print("[INFO] In EPOCH {} the loss value improved from {:.5f} to {:.5f}".format(epoch,
self.min_validation_loss, valLoss))
      self.setMinValLoss(valLoss)
      self.setCounter(0)
      self.setBestEpoch(epoch)
      torch.save(model.state_dict(), f"{models_dir}/ResNet18_state_dict_test_val_02.pt")
      self.setBestLosses(trainLoss, valLoss)

    elif valLoss > (self.min_validation_loss + self.min_delta):
      self.setCounter(self.counter + 1)
      print("[INFO] In EPOCH {} the loss value did not improve from {:.5f}. This is the {}
EPOCH in a row.".format(epoch, self.min_validation_loss, self.counter))
      if self.counter >= self.patience:
```

```
            return True
        return False

    def setCounter(self, counter_state):
        self.counter = counter_state

    def setMinValLoss(self, ValLoss):
        self.min_validation_loss = ValLoss

    def setBestLosses(self, trainLoss, valLoss):
        self.best_train = trainLoss
        self.best_val = valLoss

    def setBestEpoch(self, bestEpoch):
        self.best_epoch = bestEpoch

    def getBestTrainLoss(self):
        return self.best_train

    def getBestValLoss(self):
        return self.best_val

    def getBestEpoch(self):
        return self.best_epoch

    def saveLossesLocally(self):
        np.save(f'{models_dir}/losses_train_18test_val_02.npy', np.array(self.best_train))
        np.save(f'{models_dir}/losses_val_18test_val_02.npy', np.array(self.best_val))

    def loadLossesLocally(self):
        self.best_train = np.load(f'{models_dir}/losses_train_18test_val_02.npy')
        self.best_val = np.load(f'{models_dir}/losses_val_18test_val_02.npy')
```

7. Bucla de antrenare:

```
# defining possible resnet models
model_parameters = {'resnet18': [[64, 128, 256, 512], [2, 2, 2, 2], 1, False],
                    'resnet34': [[64, 128, 256, 512], [3, 4, 6, 3], 1, False],
                    'resnet50': [[64, 128, 256, 512], [3, 4, 6, 3], 4, True],
                    'resnet101': [[64, 128, 256, 512], [3, 4, 23, 3], 4, True],
                    'resnet152': [[64, 128, 256, 512], [3, 8, 36, 3], 4, True]}

# load from drive previous results
# previous_state = torch.load(f"{models_dir}/train_state_dict.pt")
# previous_model = torch.load(f"{models_dir}/ResNet18_state_dict_Val.pt")

# loading ResNet34
model = ResNet(model_parameters['resnet18'], 3, 21, False).to(device)
```

```
# model.load_state_dict(previous_state['model_state_dict'])

# model.load_state_dict(previous_model)

from torchvision.models import resnet18
pretrained_resnet18 = resnet18(weights='IMAGENET1K_V1')
pretrained_state_dict = pretrained_resnet18.state_dict()

# loading pretrained weights
model.load_state_dict(pretrained_state_dict, strict=False)

# initializing optimizer and lr_scheduler
optimizer = torch.optim.Adam(model.parameters(), lr=INIT_LR)
# optimizer.load_state_dict(previous_state['optimizer_state_dict'])

scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3)

# initializing loss function
lossFn = CrossEntropyLoss()

# optimizing only after a certain number of batches
accumulation_steps = 4


H = {
  "total_train_loss": [],
  "total_val_loss":[],
  "time":[]
}

# load previous loss history and last epoch
# H = previous_state['train_loss_history']
# last_epoch = previous_state['epoch']+1

print("[INFO] training the network...")
early_stopper = EarlyStopping(patience=5)
startTime = time.time()
H["time"].append(startTime)

for e in range(EPOCHS):
  # set the model to training mode
  model.train()

  total_train_loss = 0
  total_val_loss = 0

  for step, (images, gt_labels) in enumerate(trainDataLoader):
    # forward pass
    images = images.to(device)
```

```
    gt_labels = gt_labels.to(device)

    pred_labels = model(images)

    loss = lossFn(pred_labels, gt_labels) / accumulation_steps
    loss.backward()

    # we will apply weights optimization every 3 batches for more stable gradients
    # 4 batches x 6 images/batch = 24 data passes
    if step + 1 != 1:
      if (accumulation_steps % (step + 1) == 0) or (trainSteps % (step + 1) != 0 and
trainSteps == (step + 1)):
        optimizer.step()
        optimizer.zero_grad()
        # print(f"Train loss at {step+1} is:
{(train_loss+loss.cpu().detach().numpy())/step+1}")

    total_train_loss += loss.cpu().detach().numpy()


  with torch.no_grad():
    model.eval()

    for images, gt_labels in valDataLoader:
      images = images.to(device)
      gt_labels = gt_labels.to(device)

      pred_labels = model(images)

      loss = lossFn(pred_labels, gt_labels)

      total_val_loss += loss.cpu().detach().numpy()


  # calculate the average training and validation loss
  avgTrainLoss = total_train_loss / trainSteps
  avgValLoss = total_val_loss / valSteps

  # Adding another step through an epoch to the scheduler
  scheduler.step(avgValLoss)

  # Saving training and evaluation history
  H['total_train_loss'].append(avgTrainLoss)
  H['total_val_loss'].append(avgValLoss)
  H['time'].append(time.time())
  # saving current state dicts of the epoch and loss history - checkpoint
  torch.save({
    'epoch': e,
    'model_state_dict': model.state_dict(),
```

```
    'optimizer_state_dict': optimizer.state_dict(),
    'train_loss_history': H
  }, f"{models_dir}/train_state_dict_18_test_val_02.pt")

  # print the model training and validation information
  print("[INFO]: EPOCH: {}/{}...".format(e+1, EPOCHS))
  print("Train loss: {:.5f}".format(avgTrainLoss))
  print("Validation loss: {:.5f}".format(avgValLoss))
  # checking if resulting loss in evaluation improved
  if early_stopper.earlyStop((e+1), avgTrainLoss, avgValLoss, model):
    # if not improved - stop the training
    print("[INFO] Early Stopping the train process. Patience exceeded!")
    print("============================================================")
    break

  print("============================================================")

# finishing the training and calculating train time
endTime = time.time()
print("[INFO] Total time taken to train the model: {:.2f}s".format(endTime-startTime))
print("[INFO] Best loss was found in Epoch {} where the performance was {:.5f}. "
      "Model's parameters saved!".format(early_stopper.getBestEpoch(),
early_stopper.getBestValLoss()))

early_stopper.saveLossesLocally()
```

8. Etapa de testare:

```
# switching off autograd for eval
test_true_labels = []
test_pred_labels = []
H['total_test_loss'] = []

with torch.no_grad():
  # set the model in eval mode
  model.eval()

  for images, gt_labels in testDataLoader:
    images = images.to(device)
    gt_labels = gt_labels.to(device)

    pred_labels = model(images)
    loss = lossFn(pred_labels, gt_labels)
    H['total_test_loss'].append(loss.cpu().detach().tolist())
    test_true_labels.extend(gt_labels.cpu().detach().tolist())
    test_pred_labels.extend(pred_labels.cpu().detach().tolist())

  totalTestLoss = np.array(H['total_test_loss'])
```

```
  print(f"Cross Entropy on test:{np.sum(totalTestLoss)/testSteps}")
float_true = np.array(test_true_labels, dtype=float)
float_pred = np.zeros(len(test_pred_labels), dtype=float)

for i in range(len(test_pred_labels)):
  float_pred[i] = np.argmax(test_pred_labels[i])

print(float_true.shape)
print(float_pred.shape)
print(classification_report(float_true, float_pred, labels=range(0,21),
target_names=dataset.classes))

confusionMatrix = confusion_matrix(float_true, float_pred, labels=range(0,21))


displayConfusion = ConfusionMatrixDisplay(confusionMatrix, display_labels=dataset.classes)

displayConfusion.plot(xticks_rotation=90)

plt.savefig('confusion34_ext.png')

# print(float_pred)
# print(float_true)
# print(lossFn(torch.tensor(float_true), torch.tensor(float_pred))/len(float_pred))
```