Clasificarea Imaginilor din Setul de Date UC MERCED Land Use Folosind MLP și GoogLeNet

Proiect

Inteligență Computațională 2

Student
Bianca-Alexandra GHIORGHIU

Profesor
Victor Emil NEAGOE

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI

30 mai 2023

Cuprins

Introducere

Descrierea Proiectului

Scopul proiectului este de a realiza o comparație între două abordări de clasificare a imaginilor pe setul de date UC Merced, folosind rețele neuronale: MLP și transfer learning pe modelul GoogLeNet. De asemenea, se va folosi tehnica de reducere a dimensionalității, numită Principal Component Analysis (PCA), pentru a îmbunătăți performanțele modelelor.

UC Merced este un set de date de imagini aeriene ale diferitelor tipuri de terenuri, cum ar fi clădiri, câmpuri, autostrăzi și păduri. Scopul este de a antrena un model capabil să clasifice aceste imagini cu o precizie cât mai mare.

În prima parte a proiectului, se va utiliza MLP, care este o rețea neurală feedforward, pentru a clasifica imaginile din setul de date. MLP este un model simplu și flexibil, dar poate fi limitat în capacitatea sa de a extrage caracteristici complexe din imagini.

În a doua parte a proiectului, se va utiliza transfer learning pe modelul GoogLeNet, care este o rețea neurală convoluțională pre-antrenată pe setul de date ImageNet. Acest model are o arhitectură foarte complexă și poate extrage caracteristici mult mai complexe din imaginile de clasificat.

În cazul MLP-ului, se va utiliza PCA pentru a reduce dimensionalitatea datelor de antrenament și de testare. PCA este o tehnică de analiză a datelor care transformă un set de variabile correlate într-un set de variabile neliniare, ne-correlate. Această tehnică poate fi folosită pentru a reduce dimensiunea setului de date, fără a pierde prea multe informații.

Scopul final al acestui proiect este de a compara performanțele celor două modele și de a determina care abordare este mai bună pentru clasificarea imaginilor din setul de date UC Merced.

Structură

- În cadrul **contextului teoretic**, se prezintă conceptele de inteligență artificială, machine learning și deep learning, precum și algoritmi specifici, cum ar fi Principal Component Analysis, Multi-Layer Perceptron, Transfer Learning și GoogLeNet.
- În capitolul Capitolul de rezultate prezintă analiza performanțelor modelelor implementate, incluzând acuratețea, timpul de execuție și matricea de confuzie., se

prezintă setul de date utilizat și etapele implementării modelelor Multi-Layer Perceptron și GoogLeNet, precum și optimizarea lor.

– Capitolul de **rezultate** prezintă analiza performanțelor modelelor implementate, incluzând acuratețea, timpul de execuție și matricea de confuzie.

– Capitolul de **concluzii** conține o recapitulare a obiectivelor, realizări și limite ale proiectului, precum și direcții viitoare de cercetare.

Capitolul 1

Context Teoretic

1.1 Introducere

1.1.1 Inteligența Artificială

Inteligenta artificiala (IA) este o ramură a informaticii care se ocupă cu crearea de sisteme și algoritmi care pot simula abilitățile umane de a învăța, raționa, percepe și rezolva probleme. Aceasta este în mod obișnuit împărțită în trei clase:

Artificial Narrow Intelligence (ANI): se referă la sistemele AI specializate într-un singur domeniu și capabile să efectueze sarcini specifice. Aceste sisteme sunt antrenate pentru a rezolva probleme specifice și nu au capacitatea de a gândi sau de a efectua sarcini din domenii diferite.

Artificial General Intelligence (AGI): este capacitatea sistemelor AI de a realiza activități intelectuale la fel de bine sau mai bine decât un om în mai multe domenii. Aceasta este capabilă să utilizeze experiența dintr-un domeniu și să o aplice în alt domeniu sau să efectueze mai multe tipuri de sarcini cu aceeași eficiență. De asemenea, aceasta este capabilă să ia decizii și să gândească în mod autonom și să învețe din propriile greșeli.

Artificial Super Intelligence (ASI): este capacitatea sistemelor AI de a realiza activități intelectuale la un nivel superior celui uman, depășind capacitățile umane. Aceasta este capabilă să gândească, să învețe și să ia decizii cu mult mai multă precizie și eficiență decât orice ființă umană. [1]

1.1.2 Machine Learning

Învățarea automată este o ramură a inteligenței artificiale care se concentrează pe studiul metodelor care permit sistemelor să se îmbunătățească autonom din experiență [2]. Există trei componente fundamentale ale învățării automate:

Setul de date: Eșantioanele de date folosite pentru a antrena algoritmi de învățare automată.

Caracteristici: Informații esențiale, deoarece indică sistemului pe ce trebuie să se concentreze.

Algoritm: Aceeași sarcină poate fi abordată cu algoritmi diferiți. Criteriile privind precizia, calitatea rezultatelor și puterea de procesare depind de algoritmul ales. De asemenea, este de remarcat faptul că, ocazional, combinarea a doi sau mai mulți algoritmi poate îmbunătăți performanța (învățare în ansamblu).

Cele cinci categorii principale de învățare automată sunt învățarea supervizată, învățarea nesupervizată, învățarea semisupervizată, învățarea prin recompensă și reducerea dimensionalității.

1.1.3 Deep Learning

Învățarea profundă este un subset al învățării automate care utilizează rețele neuronale artificiale cu mai multe straturi pentru a învăța și a face predicții sau alegeri din cantități mari de date. Pe măsură ce sunt expuse la mai multe date, rețelele neuronale sunt antrenate să identifice tipare în date și să facă predicții din ce în ce mai precise. Modelele de învățare profundă au fost utilizate în mod eficient pentru o gamă largă de sarcini, inclusiv recunoașterea imaginilor și a vorbirii, procesarea limbajului natural și luarea autonomă a deciziilor. Termenul "profund" face aluzie la numărul mare de straturi ale rețelei neuronale, care o ajută să învețe reprezentări complexe ale datelor.

1.2 Principal Component Analysis

Procesul de transformare liniară sau neliniară a spațiului original al observațiilor într-un spațiu cu dimensiuni reduse în care se execută algoritmul de clasificare a datelor este definit ca selecție de caracteristici. Unele metode de clasificare, explorare sau analiză care sunt eficiente într-un mediu cu dimensiuni reduse pot deveni nepractice într-un spațiu cu dimensiuni ridicate. În plus, deoarece deseori sunt necesare decizii în timp real, minimizarea volumului de calcul este esențială [3].

Analiza componentelor principale (PCA), cunoscută sub numele de Transformarea Karhunen-Loève, este o abordare statistică eficientă pentru selectarea caracteristicilor care a devenit un standard în aplicațiile de data mining în ultimele decenii [3]. Este o tehnică de reducere a dimensionalității setului de date, de îmbunătățire a interpretabilității și de minimizare a pierderilor de informații. Acest lucru se realizează prin transformarea liniară a datelor într-un nou sistem de coordonate în care (cea mai mare parte a) varianței din date poate fi reprezentată cu mai puține dimensiuni decât datele originale [4].

Știind că avem un set n-dimensional de vectori $X_1, \ldots, X_L$, pașii algoritmului PCA sunt următorii [3]:

Etapa 1: Se calculează matricea de covarianță pentru caracteristicile din setul de date

$$\Sigma_X = \frac{1}{L} \sum_{i=1}^{L} (X_i - \mu_x)(X_i - \mu_x)^t \tag{1.1}$$

unde

$$\mu_x = \frac{1}{L}\sum_{i=1}^{L} X_i \tag{1.2}$$

Etapa 2: Se calculează valorile proprii pentru matricea de covarianță.

Valorile proprii λ_i, $i = 1, \ldots, n$ ale matricei Σ_X, se determină ca soluție a ecuației:

$$|\Sigma_X - \lambda \cdot I_n| = 0 \tag{1.3}$$

unde I_n este matricea identitate n-dimensională.

Rădăcinile ecuației 1.3, care reprezintă valorile proprii, sunt ordonate în ordine descrescătoare, cele mai mari m valori proprii fiind reținute.

Etapa 3: Se calculează vectorii proprii pentru matricea de covarianță.

Vectorii proprii sunt determinați prin rezolvarea următorului sistem de ecuații:

$$\begin{cases} \Sigma_X \cdot \Phi_i = \lambda_i \cdot \Phi_i \\ \Phi_i^t \cdot \Phi_i = 1 \end{cases}$$

unde $\Phi_1, \ldots, \Phi_m, \Phi_{m+1}, \ldots, \Phi_n$ sunt vectorii proprii ai lui Σ_X, care corespund valorilor proprii în ordine descrescătoare

Etapa 4: Se construiește matricea de vectori proprii K

$$K = (\Phi_1 | \ldots | \Phi_n)^t \tag{1.4}$$

Etapa 5: Se construiește matricea PCA

$$PCA = (\Phi_1 | \ldots | \Phi_m)^t \tag{1.5}$$

unde m este numărul de caracteristici care se dorește a fi păstrate după aplicarea PCA.

Etapa 6: Se transformă matricea originală

Prin aplicarea transformării PCA se obțin vectori cu dimensionalitate redusă în spațiul m-dimensional (unde $m < n$).

$$Z_i = PCA \cdot X_i \tag{1.6}$$

1.3 Multi-Layer Perceptron

1.3.1 Arhitectura

Un perceptron cu mai multe straturi (MLP) este o rețea neuronală artificială, care este utilizată pentru a rezolva probleme de clasificare și regresie. MLP constă din cel puțin

trei straturi: un strat de intrare, unul sau mai multe straturi ascunse și un strat de ieșire [5], așa cum se poate observa în figura de mai jos.

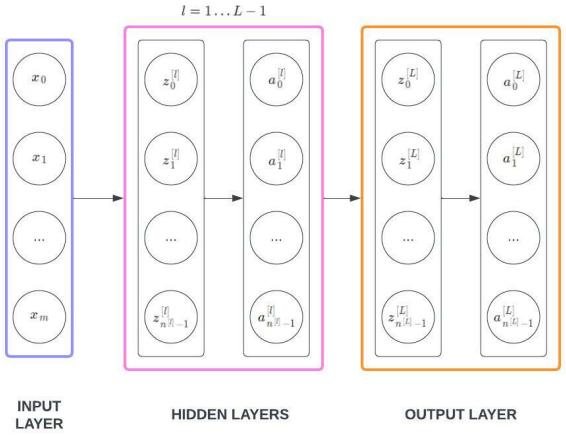

Figura 1.1: Arhitectura MLP-ului

MLP este compus din trei sau mai multe straturi: intrare, ascuns și ieșire. Straturile de intrare primesc datele de intrare, cum ar fi o imagine a unei cifre scrise de mână. Fiecare neuron din stratul de intrare primește valorile pixelilor și transmite semnalul de ieșire către următorul strat. Straturile ascunse extrag caracteristicile semnificative ale datelor de intrare prin transformarea semnalelor de la straturile anterioare în semnale de ieșire utilizând o funcție de activare. Numărul de neuroni în straturile ascunse poate varia. Stratul de ieșire primește semnalele de la ultimul strat ascuns și calculează ieșirea finală a rețelei neuronale.

1.3.2 Forward Propagation

Este important să se definească mai întâi câteva notații înainte de a trece la explicarea algoritmului de forward propagation:

- Indicele [l] indică o cantitate asociată cu stratul l.
- Indicele (i) corespunde intrărilor de strat.
- Indicele i specifică numărul neuronului din cadrul stratului.

În acest caz, potențialul de activare arată astfel:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \tag{1.7}$$

unde

$$W^{[l]} = \begin{bmatrix} w_1^{[l](1)} & w_1^{[l](2)} & \dots & w_1^{[l](n^{[l-1]})} \\ w_2^{[l](1)} & w_2^{[l](2)} & \dots & w_2^{[l](n^{[l-1]})} \\ \vdots & \vdots & & \vdots \\ w_{n^{[l]}}^{[l](1)} & w_{n^{[l]}}^{[l](2)} & \dots & w_{n^{[l]}}^{[l](n^{[l-1]})} \end{bmatrix} \quad A^{[l-1]} = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \dots & a_1^{[l](n^{[l-1]})} \\ a_2^{[l](1)} & a_2^{[l](2)} & \dots & a_2^{[l](n^{[l-1]})} \\ \vdots & \vdots & & \vdots \\ a_m^{[l](1)} & a_m^{[l](2)} & \dots & a_m^{[l](n^{[l-1]})} \end{bmatrix} \quad b^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}$$

Aplicând $Z^{[l]}$ la funcția de activare $f^{[l]}$ se obține o ieșire care poate fi aplicată la un strat următor:

$$A^{[l]} = f^{[l]}(Z^{[l]}) \tag{1.8}$$

1.3.3 Backpropagation

Conform articolului original care descrie backpropagation, scopul rețelei neuronale este de a descoperi o combinație de ponderi care să asigure că, pentru fiecare vector de intrare, vectorul de ieșire al rețelei este cât mai aproape posibil de vectorul de ieșire țintă [6]. Prima etapă în determinarea ponderilor adecvate constă în definirea unei funcții de cost, care măsoară cât de incorect identifică modelul relația dintre intrare și ieșire. Aceasta indică cât de slab funcționează modelul.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}(i), y(i)) \tag{1.9}$$

unde L este funcția de pierdere aleasă în funcție de aplicație și m este numărul de eșantioane din setul de date de antrenament.

Funcția de cost este, în mod ideal, o funcție convexă, așa cum este reprezentată în 1.2. Problema optimizării constă în minimizarea acestei funcții prin determinarea ponderilor optime cu ajutorul algoritmului gradient descent. Ponderile și biasurile sunt inițializate la numere aleatoare mici, ceea ce determină algoritmul să pornească dintr-un punct arbitrar al funcției din figura 1.2. Apoi ne deplasăm iterativ în jos în direcția celei mai abrupte pante, actualizând valorile w și b în conformitate cu următoarele formule:

$$w = w - \alpha \frac{\partial J(w, b)}{\partial w} \tag{1.10}$$

$$b = b - \alpha \frac{\partial J(w, b)}{\partial w} \tag{1.11}$$

Termenul alfa reprezintă rata de învățare și indică mărimea pasului de gradient al fiecărei iterații. O rată de învățare prea mare poate duce la convergența rapidă a modelului către o soluție slabă, în timp ce o rată de învățare prea mică poate face ca procedura să dureze foarte mult timp.

Figura 1.3 este utilă pentru a înțelege mai bine cum funcționează algoritmul gradient descent. Atunci când panta este pozitivă, componenta derivată este pozitivă și, datorită semnului negativ înaintea termenului de gradient, mergem în cealaltă direcție, spre un w descrescător.

În funcție de frecvența cu care sunt actualizate ponderile și biasurile, se disting mai multe tipuri de gradient descent [9]:

– **Batch Gradient Descent:** După calcularea gradientului funcției de cost $J(w, b)$ în raport cu întregul set de instruire, parametrii w și b sunt modificați.

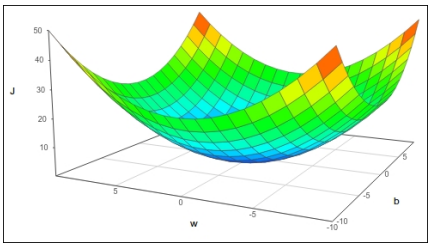

Figura 1.2: Funcția de Cost [7]

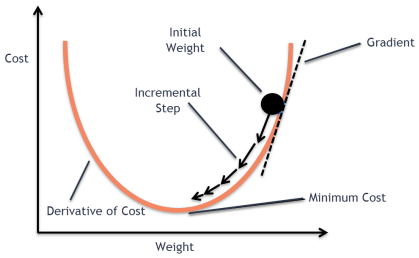

Figura 1.3: Gradient Descent [8]

– **Mini Batch Gradient Descent:** După calcularea gradientului funcției de cost $J(w, b)$ în raport cu o parte din setul de instruire, parametrii w și b sunt modificați.

– **Stochastic Gradient Descent:** După calcularea gradientului funcției de cost $J(w, b)$ în raport cu un singur exemplu din setul de instruire, se modifică parametrii w și b.

1.4 Transfer Learning

Transfer learning este o tehnică de învățare automată care permite unui model antrenat pentru o sarcină să fie utilizat pentru o altă sarcină similară, dar diferită. În loc să înceapă procesul de învățare de la zero, transferul de învățare utilizează cunoștințele și caracteristicile învățate de un model pre-antrenat pe un set de date mare, pe care le poate aplica ulterior pe un set de date nou, mai mic [10].

Există două abordări principale: extragerea de caracteristici și fine-tuning. În extragerea de caracteristici, ponderile modelului pre-antrenat sunt blocate, iar doar stratul de ieșire este înlocuit cu unul nou care corespunde cerințelor noii sarcini. Această abordare utilizează capacitatea modelului pre-antrenat de a extrage caracteristici semnificative din date și le aplică pe noua sarcină. Noul strat de ieșire este apoi antrenat pe noul set de date, în timp ce restul modelului pre-antrenat rămâne neschimbat [11].

Pe de altă parte, fine-tuning-ul implică deblocarea unor sau a tuturor straturilor modelului pre-antrenat și antrenarea întregului model pe noul set de date. Această abordare permite modelului pre-antrenat să se adapteze la noua sarcină prin actualizarea greutăților sale pe baza noilor date, menținând totodată caracteristicile învățate în timpul pre-antrenamentului [11].

1.5 GoogLeNet

GoogLeNet, cunoscut și sub numele de Inception-v1, este o arhitectură de rețea neuronală convoluțională profundă (CNN) concepută pentru clasificarea imaginilor și sarcini de detecție a obiectelor. A fost dezvoltat de cercetătorii de la Google și introdus în lucrarea "Going Deeper with Convolutions" în 2014. GoogLeNet a câștigat ImageNet Large Scale Visual Recognition Challenge (ILSVRC) în acel an, stabilind noi standarde pentru precizie și eficiență computațională [12].

Arhitectura GoogLeNet este caracterizată de următoarele caracteristici principale:

1. **Module Inception:** Blocurile de construcție de bază ale GoogLeNet sunt modulele Inception, care sunt concepute pentru a permite rețelei să învețe caracteristici la scară și nivel multiplu în mod eficient. Fiecare modul Inception conține mai multe straturi convoluționale paralele cu dimensiuni diferite ale filtrelor, urmate de un strat de max-pooling. Ieșirile acestor straturi sunt apoi concatenate, permițând în esență rețelei să învețe caracteristici la mai multe scale simultan.

2. **22 de straturi:** GoogLeNet este o rețea adâncă cu 22 de straturi, inclusiv straturi convoluționale, de pooling și complet conectate. Această adâncime permite rețelei să învețe caracteristici complexe și ierarhice, care contribuie la performanța sa ridicată în sarcinile de clasificare a imaginilor.

3. **Global average pooling:** În loc să utilizeze straturi complet conectate înainte de stratul final de clasificare, GoogLeNet folosește global average pooling. Această strategie reduce numărul de parametri din rețea și atenuează riscul de overfitting. Stratul de global average pooling calculează valoarea medie pentru fiecare hartă de caracteristici, iar aceste valori sunt apoi introduse direct în stratul softmax final pentru clasificare.

4. **Clasificatori auxiliari:** Pentru a aborda problema de vanishing gradients, care poate apărea în rețelele adânci, GoogLeNet introduce clasificatori auxiliari. Aceștia sunt adăugați la straturile intermediare ale rețelei și oferă supraveghere suplimentară în timpul antrenării, ajutând gradientul să curgă mai eficient prin rețea. Clasificatorii auxiliari folosesc o arhitectură CNN mai mică, urmată de pooling mediu, straturi complet conectate și un strat softmax. Pierderea de la acești clasificatori este combinată cu pierderea clasificatorului principal în timpul antrenării, dar nu este utilizată în timpul inferenței.

5. **Utilizarea eficientă a parametrilor:** GoogLeNet este conceput pentru a fi eficient din punct de vedere computațional, ceea ce îl face potrivit pentru implementare pe o varietate de dispozitive, inclusiv telefoane mobile și sisteme încorporate. Modulele Inception, pooling-ul și utilizarea redusă a straturilor complet conectate contribuie la un număr semnificativ mai mic de parametri în comparație cu alte CNN-uri adânci de la vremea respectivă, cum ar fi AlexNet și VGG.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figura 1.4: Arhitectura GoogLeNet

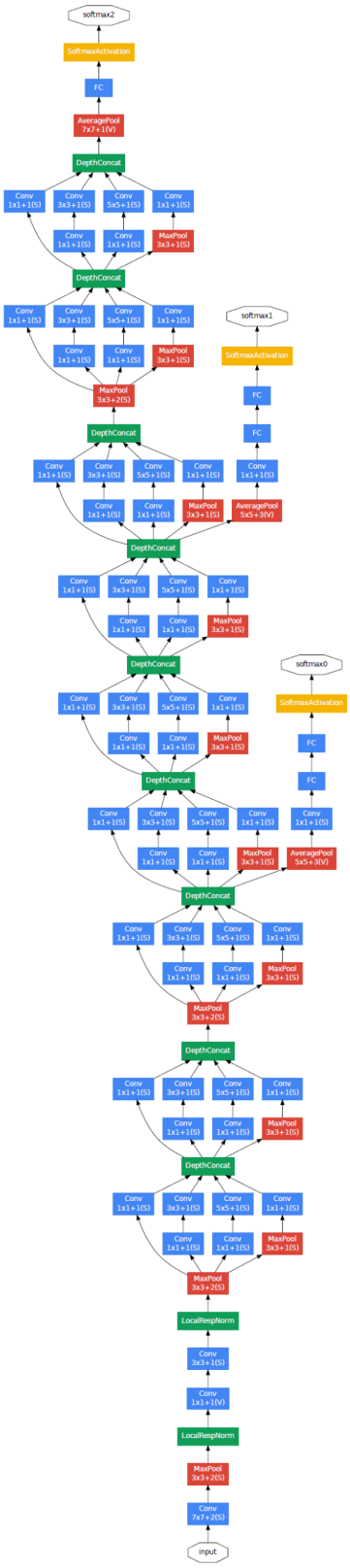

Figura 1.5: Arhitectura GoogLeNet

Capitolul 2

Descrierea Implementării

2.1 Setul de Date

Setul de date UC Merced este un set de date creat de Universitatea din California, Merced. Acesta este conceput pentru sarcini de clasificare a utilizării terenurilor și a acoperirii terenurilor. Setul de date conține 2.100 de imagini aeriene cu rezoluție înaltă, cu o rezoluție de 0.3 metri per pixel. Aceste imagini sunt împărțite în 21 de clase distincte de utilizare a terenurilor, cum ar fi agricol, avion, teren de baseball, plajă, clădiri etc [13].

Fiecare dintre cele 21 de clase conține 100 de imagini, oferind un set de date echilibrat pentru sarcinile de clasificare. De la lansarea sa, setul de date UC Merced a fost folosit pe scară largă în comunitățile de cercetare în teledetecție și viziune computerizată pentru sarcini precum clasificarea imaginilor, segmentarea și detectarea obiectelor [13]. Câteva exemple din setul de date se pot observa în imaginea de mai jos.

Figura 2.1: Setul de Date UC Merced

2.2 Etapele Implementării

2.2.1 MLP

– După încărcarea datelor, acestea sunt împărțite într-un set de antrenare, testare și validare. De asemenea, datele sunt procesate astfel încât clasele să fie distribuite în mod uniform în fiecare set.

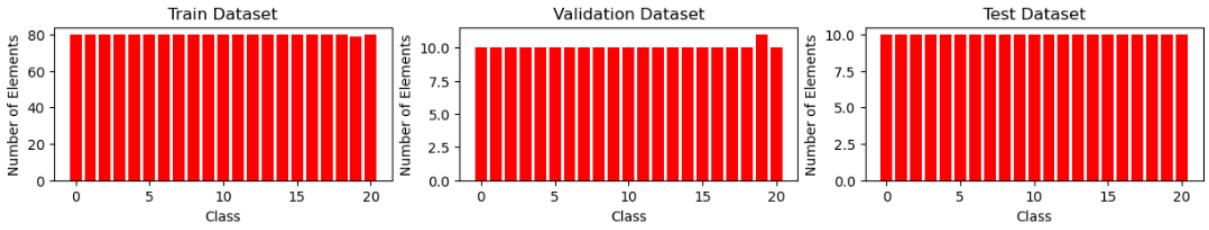

Figura 2.2: Distribuția Claselor

– Imaginile sunt normalizate, iar etichetele lor sunt convertite în codificări one-hot.

– Se aplică PCA (Principal Component Analysis) asupra datelor pentru a reduce dimensionalitatea. S-a constatat că utilizarea a 100 de componente principale reduce dimensionalitatea și păstrează aproximativ 70% din informația utilă.

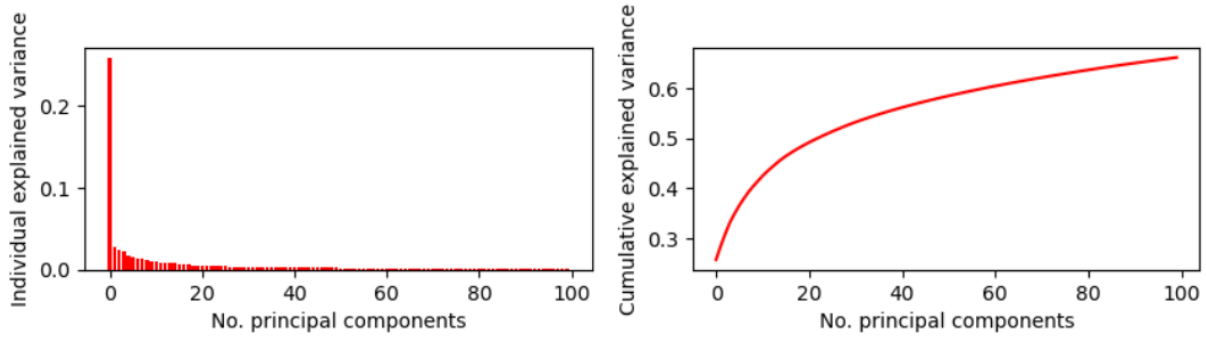

Figura 2.3: Varianța Individuală și Cumulativă

– Parametrii modelului MLP sunt optimizați folosind Optuna.

– Pe baza celor mai buni hiperparametri găsiți, modelul MLP este antrenat pentru 10-50 de epoci. De asemenea, se măsoară timpul de execuție al antrenării.

– Se plotează curbele de învățare pentru pierdere (loss) și acuratețe (accuracy) pe seturile de antrenare și validare, pentru a evalua procesul de antrenare.

– Se calculează acuratețea pentru setul de testare și se măsoară timpul de execuție necesar pentru testare.

– Se generează și se afișează matricile de confuzie pentru seturile de antrenare, testare și validare.

2.2.2 GoogLeNet

– După încărcarea datelor, acestea sunt împărțite într-un set de antrenare, testare și validare. De asemenea, datele sunt procesate astfel încât clasele să fie distribuite în

mod uniform în fiecare set.

- Unele transformări sunt aplicate asupra datelor: redimensionarea la dimensiunea de 224x224 și normalizarea. În plus, se utilizează și augmentarea datelor, adăugând inversări orizontale aleatorii și rotații aleatorii cu o probabilitate de 10
- Hiperparametrii utilizați pentru ajustarea fină pe GoogLeNet sunt optimizați folosind Optuna.
- Cu cei mai buni hiperparametri găsiți, se efectuează fine-tuning pe GoogLeNet pentru 10-50 de epoci. De asemenea, se măsoară timpul de execuție pentru acest proces.
- Se plotează curbele de învățare pentru pierdere (loss) și acuratețe (accuracy) pe seturile de antrenare și validare, pentru a evalua procesul de antrenare.
- Se calculează acuratețea pentru setul de testare și se măsoară timpul de execuție necesar pentru testare.
- Se generează și se afișează matricile de confuzie pentru seturile de antrenare, testare și validare.

2.3 Descrierea Modelelor

2.3.1 MLP

- **Arhitectura:** Modelul este o rețea neuronală profundă de tip feed-forward cu șase straturi dense. Fiecare strat are 1661 de unități, folosește funcția de activare ReLU, batch normalization si dropout cu o rată de 0.20.
- **Optimizator:** Modelul utilizează optimizatorul de tip Stochastic Gradient Descent (SGD) cu o rată de învățare de 0.007 și un momentum de 0.74.
- **Funcția de Pierdere și Metrici:** Se utilizează categorical cross-entropy ca funcție de pierdere și acuratețea ca metrică de evaluare.
- **Mărimea Lotului și Epoci:** Modelul este antrenat cu o mărime a lotului de antrenare de 53 pentru un total de 10-50 epoci.

2.3.2 GoogLeNet

- **Arhitectura:** Modelul folosit este GoogleNet pre-antrenat, care este o rețea neuronală convoluțională profundă, cunoscută pentru complexitatea și adâncimea sa mare. Ultimul strat complet conectat fully-connected al modelului este modificat pentru a se potrivi cu numărul de clase necesare.
- **Optimizator**: Modelul folosește optimizatorul Stochastic Gradient Descent (SGD) cu o rată de învățare de aproximativ 0.0074, weight decay de aproximativ 0.00088 și momentum de 0.74.
- **Funcția de Pierdere și Metrici:** Modelul folosește categorical cross-entropy, care este comună pentru sarcinile de clasificare multi-clasă. Performanța modelului este evaluată folosind acuratețea ca metrică.

– **Mărimea Lotului și Epoci:** Mărimea lotului este de 32. Modelul este antrenat pentru un total de 10-50 epoci.

2.4 Optimizarea Modelelor

2.4.1 MLP

Opimizarea Modelului

```
def objective(trial):
    # Define the hyperparameters to be optimized
    activation_function = trial.suggest_categorical("activation_function
   ", ['relu', 'sigmoid', 'tanh'])
    units = trial.suggest_int("units", 2, 2048, log=True)
    dropout_rate = trial.suggest_uniform("dropout_rate", 0.1, 0.7)

    # Define the model architecture
    model = Sequential()
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(units, activation=activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))

    # Define the optimizer
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "
   RMSprop","SGD"])
    learning_rate = trial.suggest_loguniform("learning_rate", 1e-5, 1e
   -1)

    # Define Adam pramaeters
    beta_1 = trial.suggest_float("beta_1", 0.8, 0.999)
    beta_2 = trial.suggest_float("beta_2", 0.99, 0.9999)
    epsilon = trial.suggest_float("epsilon", 1e-11, 1e-5)

    # Define RMSprop/SGD praameters
    momentum = trial.suggest_uniform('momentum', 0.0, 1.0)

    if optimizer_name == "Adam":
        optimizer = Adam(learning_rate=learning_rate,
                         beta_1=beta_1,
                         beta_2=beta_2,
```

```
                        epsilon=epsilon)
    elif optimizer_name == "RMSprop":
        optimizer = RMSprop(learning_rate=learning_rate,
                            momentum=momentum)
    elif optimizer_name == "SGD":
        optimizer = SGD(learning_rate=learning_rate,
                        momentum=momentum)


    # Compile the model
    model.compile(optimizer=optimizer, loss="categorical_crossentropy",
   metrics=["accuracy"])

    # Train the model
    batch_size = trial.suggest_int('batch_size',1, 64, log=True)
    callback = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',
   patience=10)
    history = model.fit(X_train, y_train, batch_size=batch_size,
                        validation_data=(X_val, y_val),
                        callbacks=[callback],
                        epochs=100)

    # Return the validation loss as the objective value to be minimized
    return history.history['val_accuracy'][-1]

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=50)
```

Hyperparametrii Optimi

- Functie de activare: ReLU
- Numar de neuroni: 1661
- Dropout rate: 0.20446990104038432
- Algoritm de optimizare: SGD
- Learning rate: 0.0071465808796697524
- β_1: 0.9770613118775386
- β_2: 0.9938113815665985
- ϵ: 8.563672685161736e-06
- Momentum: 0.7428620688789899
- Batch size: 53

2.4.2 GoogLeNet

Optimizarea Modelului

```
def objective(trial):
    # load pre-trained GoogLeNet model
    model = models.googlenet(pretrained=True)
    model.to(device)
```

```
    # replace last layer to match the number of classes in UC Merced
dataset
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    model.to(device)

    # set hyperparameters for Optuna
    learning_rate = trial.suggest_float('lr', 1e-5, 1e-2, log=True)
    weight_decay = trial.suggest_float('weight_decay', 1e-5, 1e-1, log=
True)
    momentum = trial.suggest_float('momentum', 0.5, 0.999)
    step_size = trial.suggest_int('step_size', 1, 30)
    gamma = trial.suggest_float('gamma', 0.1, 1.0)

    # set optimizer and loss function
    optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                          weight_decay=weight_decay, momentum=momentum)
    criterion = nn.CrossEntropyLoss()

    # create learning rate scheduler
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size
,
                                          gamma=gamma)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for train_images, train_labels in train_loader:
            train_images, train_labels = train_images.to(device),
train_labels.to(device)
            optimizer.zero_grad()
            train_predictions = model(train_images)
            loss = criterion(train_predictions, train_labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # update learning rate
        scheduler.step()

        model.eval()
        val_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for val_images, val_labels in val_loader:
                val_images, val_labels = val_images.to(device),
val_labels.to(device)
                val_predictions = model(val_images)
                loss = criterion(val_predictions, val_labels)
                val_loss += loss.item()
                _, predicted = torch.max(val_predictions.data, 1)
                total += val_labels.size(0)
                correct += (predicted == val_labels).sum().item()

        trial.report(val_loss/len(val_loader), epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()
```

```
    return val_loss/len(val_loader)

study = optuna.create_study(direction='minimize',
                            pruner=optuna.pruners.MedianPruner(
    n_warmup_steps=10))
study.optimize(objective, n_trials=5, timeout=7200)
```

Hyperparametrii Optimi

- Learning rate: 0.0074391316898536905
- Weight decay: 0.0008783418521336298
- Momentum: 0.7426637487499734
- Step size: 23
- γ: 0.5136244681640415

Capitolul 3

Rezultate

3.1 Acurateţe şi Timp de Execuţie

În tabelele de mai jos se poate observa acurateţea pe setul de test, timpul de antrenare şi cel de testare, atât pentru modelul MLP, cât şi pentru cel GoogLeNet pre-antrenat, în cazul impărţirii setului de date în 70-30, 80-20, 90-10 şi 10-50 epoci.

Tabela 3.1: 10 epoci

Split	Arhitectura	Acurateţe Test	Timp Antrenare (s)	Timp Test (s)
70:15:15	MLP	37.13%	6.84	0.12
	GoogLeNet	98.10%	99.44	1.21
80:10:10	MLP	45.87%	7.24	0.13
	GoogLeNet	98.10%	93.35	0.91
90:5:5	MLP	39.88%	8.95	0.10
	GoogLeNet	99.05%	110.00	0.80

Tabela 3.2: 20 epoci

Split	Arhitectura	Acurateţe Test	Timp Antrenare (s)	Timp Test (s)
70:15:15	MLP	39.12%	10.96	0.20
	GoogLeNet	98.73%	199.46	1.30
80:10:10	MLP	49.35%	11.00	0.17
	GoogLeNet	98.10%	186.15	0.78
90:5:5	MLP	41.54%	11.97	0.13
	GoogLeNet	98.10%	201.82	0.75

Tabela 3.3: 30 epoci

Split	Arhitectura	Acuratețe Test	Timp Antrenare (s)	Timp Test (s)
70:15:15	MLP	41.48%	13.33	0.19
	GoogLeNet	99.05%	261.72	1.29
80:10:10	MLP	52.89%	14.75	0.15
	GoogLeNet	97.89%	281.14	0.91
90:5:5	MLP	45.98%	16.28	0.11
	GoogLeNet	98.10%	307.10	0.82

Tabela 3.4: 40 epoci

Split	Arhitectura	Acuratețe Test	Timp Antrenare (s)	Timp Test (s)
70:15:15	MLP	39.74%	19.71	0.19
	GoogLeNet	98.73%	344.18	1.27
80:10:10	MLP	50.01%	20.13	0.21
	GoogLeNet	97.62%	372.27	1.00
90:5:5	MLP	42.10%	20.20	0.19
	GoogLeNet	99.05%	403.86	0.93

Tabela 3.5: 50 epoci

Split	Arhitectura	Acuratețe Test	Timp Antrenare (s)	Timp Test (s)
70:15:15	MLP	43.05%	21.14	0.18
	GoogLeNet	98.10%	427.90	1.33
80:10:10	MLP	50.58%	22.06	0.16
	GoogLeNet	99.05%	537	0.98
90:5:5	MLP	44.12%	25.22	0.15
	GoogLeNet	99.53%	504.52	0.76

În figurile de mai jos se poate observa comparația pentru acuratețea modelelor MLP și GoogLeNet pre-antrenat pentru împărțirea setului de date în proporții de 70-30, 80-20 și 90-10, pe parcursul a 10-50 epoci.

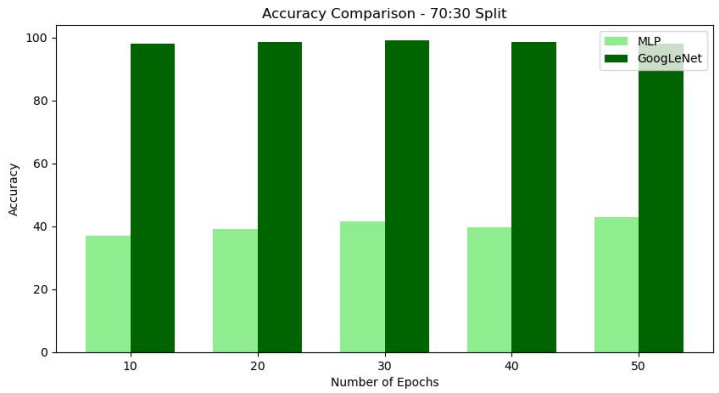

Figura 3.1: Acuratețe 70% - 30%

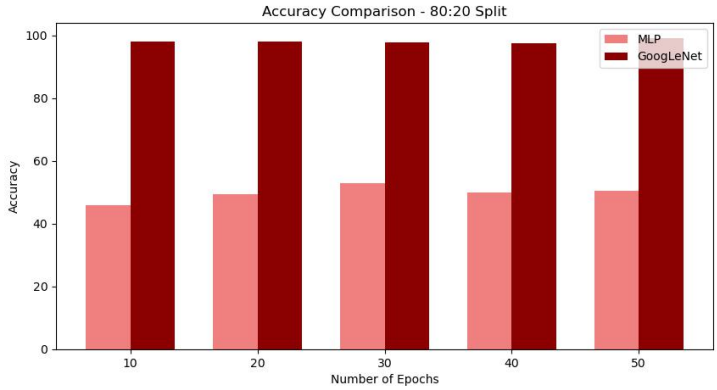

Figura 3.2: Accuratețe 80% - 20%

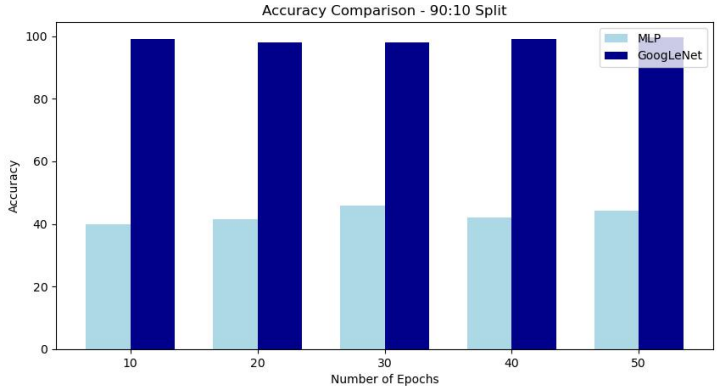

Figura 3.3: Accuratețe 90% - 10%

În figurile de mai jos se poate observa comparația pentru timpul de antrenare a modelelor MLP și GoogLeNet pre-antrenat pentru împărțirea setului de date în proporții de 70-30, 80-20 și 90-10, pe parcursul a 10-50 epoci.

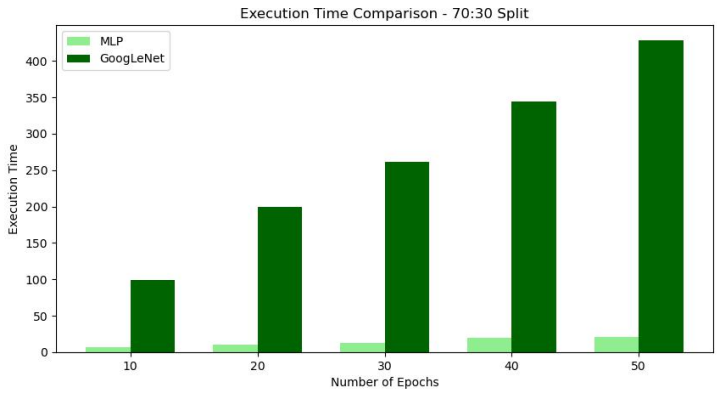

Figura 3.4: Timp Execuție 70% - 30%

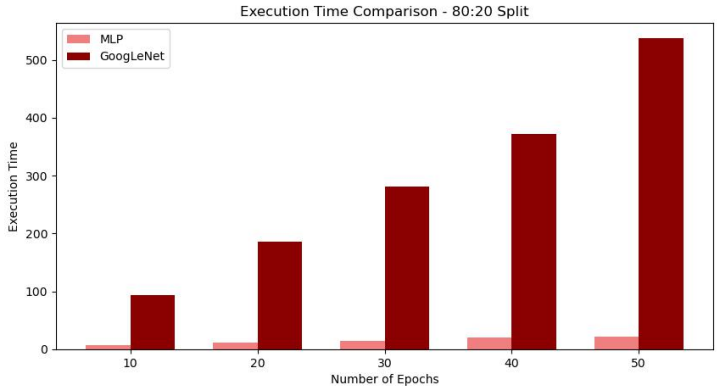

Figura 3.5: Timp Execuție 80% - 20%

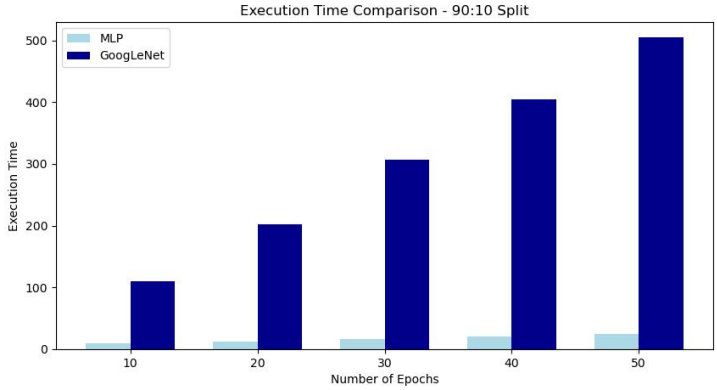

Figura 3.6: Timp Execuție 90% - 10%

În figurile de mai jos se poate observa comparația pentru timpul de testare modelelor MLP și GoogLeNet pre-antrenat pentru împărțirea setului de date în proporții de 70-30, 80-20 și 90-10, pe parcursul a 10-50 epoci.

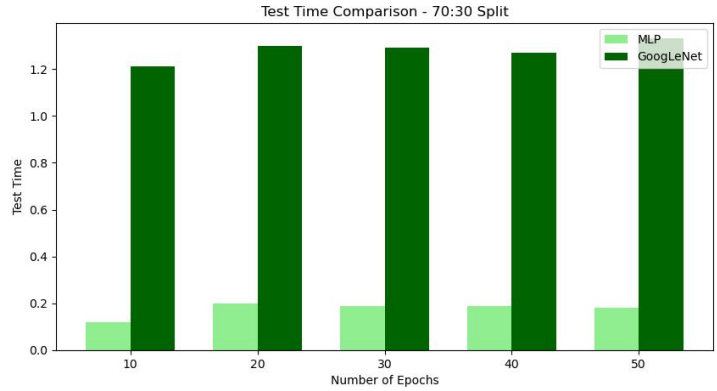

Figura 3.7: Timp Test 70% - 30%

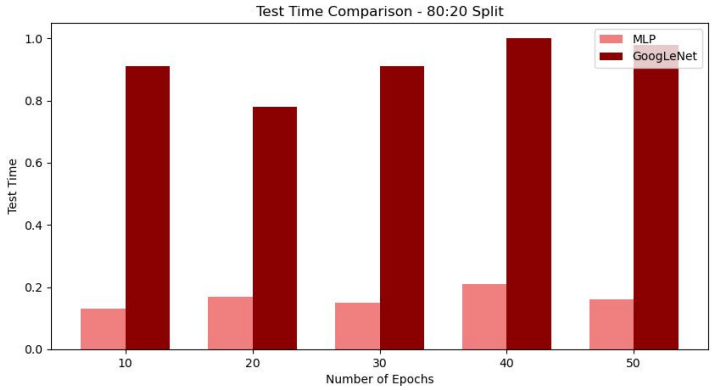

Figura 3.8: Timp Test 80% - 20%

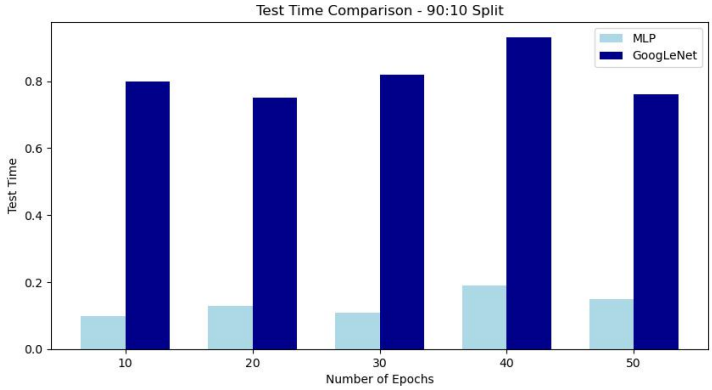

Figura 3.9: Timp Test 90% - 10%

3.2 Matricea de Confuzie

3.2.1 MLP

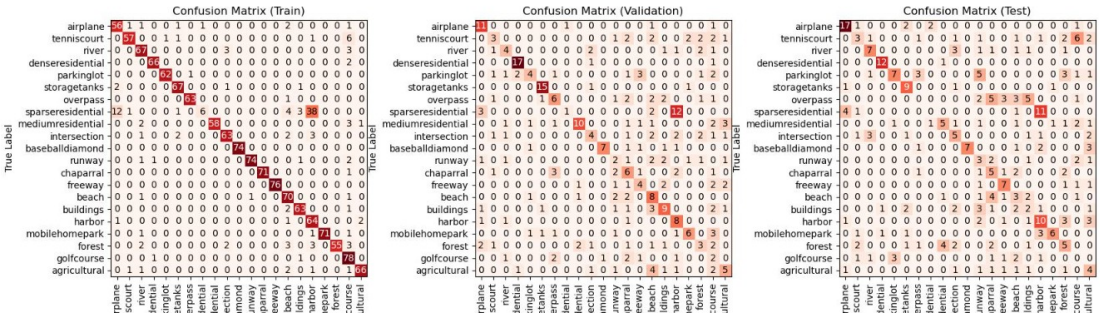

Figura 3.10: MLP, 70:30, 10 epoci

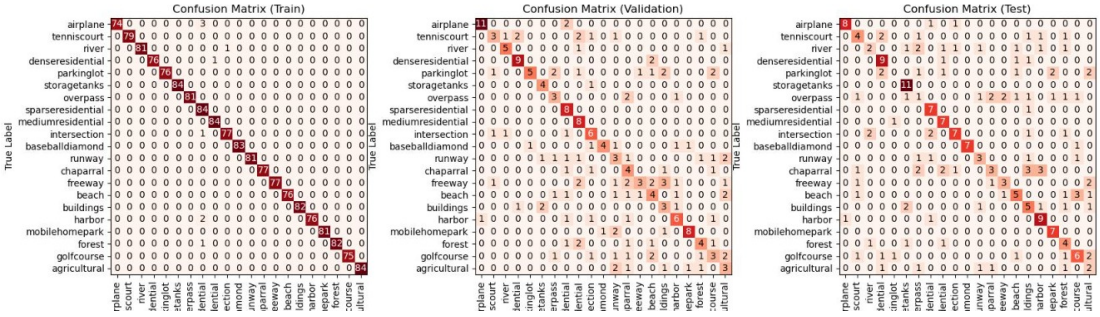

Figura 3.11: MLP, 80:20, 30 epoci

3.2.2 GoogLeNet

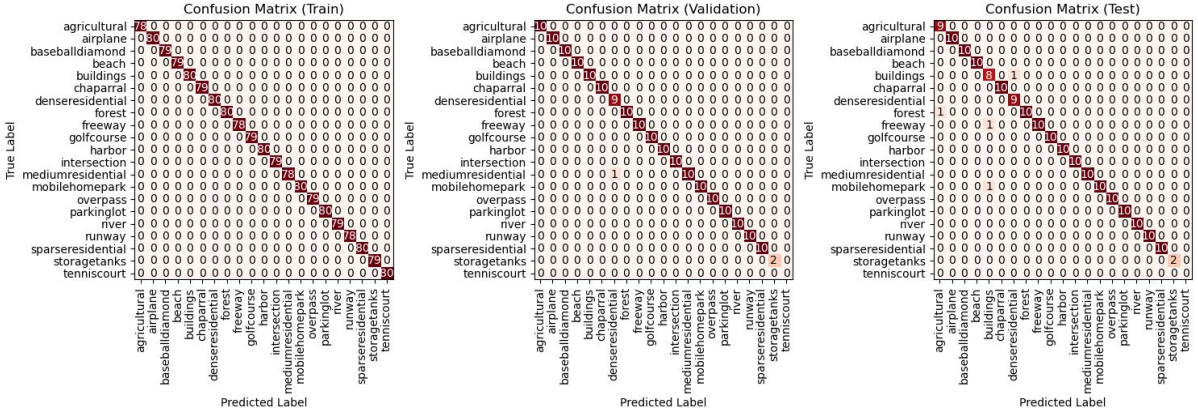

Figura 3.12: GoogLeNet, 80:20, 40 epoci

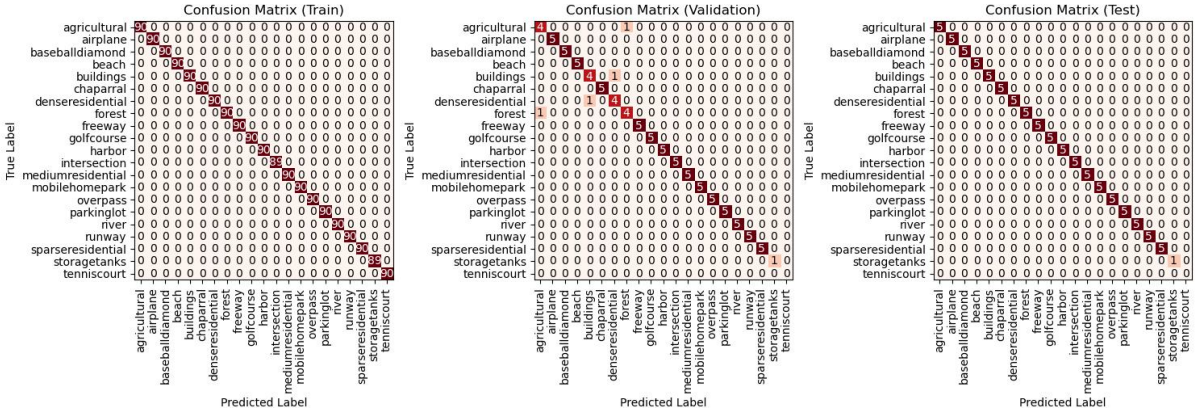

Figura 3.13: GoogLeNet, 90:10, 50 epoci

Capitolul 4

Concluzii

1. **Dimensiunile intrării: MLP** sunt de obicei limitate la **manipularea vectorilor de intrare unidimensionali**, în timp ce **GoogLeNet poate gestiona intrări de dimensiuni variabile**, datorită straturilor sale convoluționale.
2. **Complexitate:** GoogLeNet, fiind o rețea neurală convoluțională adâncă, are o arhitectură mai complexă în comparație cu MLP (**62.3M parametrii pentru GoogLeNet și 14M parametrii în cazul MLP**). Această arhitectura îi permite rețelei GoogLeNet să învețe reprezentări complexe și să obțină o performanță mai bună în sarcinile de clasificare, mai ales pentru seturi de date și intrări de dimensiuni mari, cum ar fi imaginile.
3. **Învățarea caracteristicilor:** Straturile convoluționale ale **GoogLeNet** permit **extragerea automată a caracteristicilor relevante din datele de intrare**, în timp ce **MLP necesită prelucrare manuală a caracteristicilor** pentru a obține rezultate similare.
4. **Eficiența computațională:** Datorită structurii sale mai complexe, GoogLeNet necesită în general mai multe resurse computaționale și timp de instruire mai lung decât MLP. Cei mai lungi timpi de instruire sunt pentru o împărțire **90:10 și 50 de epoci, GoogLeNet având nevoie de 504,52 secunde, iar MLP de 25,22 secunde**. După cum se poate observa, fine-tuning-ul pe GoogleNet durează de aproximativ 20 de ori mai mult decât antrenarea MLP de la început.
5. **Overfitting:** Arhitectura **GoogLeNet**, care include **dropout și clasificatorii auxiliari**, o face mai rezistentă la overfitting în comparație cu MLP.
6. **Acuratețe: MLP** a obținut cea mai mare acuratețe de **52,89%** după ce a fost antrenat timp de **30 de epoci**, utilizând o **împărțire 80:20 pentru setul de date**, în timp ce **GoogleNet** a obținut o acuratețe aproape perfectă de **99.53%** atunci când a fost antrenat timp de **50 de epoci** și utilizând o **împărțire 90:10 pentru setul de date**.
7. **Aplicabilitate:** Deși **GoogLeNet** este **potrivit în special pentru sarcinile de clasificare a imaginilor**, este posibil să nu fie alegerea optimă pentru toate tipurile de probleme de clasificare. Pe de altă parte, **MLP sunt mai generali și pot fi aplicați la o gamă mai largă de sarcini de clasificare**, deși s-ar putea să nu aibă aceeași performanță pe date complexe și de dimensiuni înalte.

Bibliografie

[1] Russell S. J. *Artificial Intelligence A Modern Approach*. Pearson Education, Inc., 2010.

[2] Mitchell T.M. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[3] Neagoe V. Computational Intelligence 1 Laboratory Platforms. 2022.

[4] Principal Component Analysis. `https://en.wikipedia.org/wiki/Principal_component_analysis`. Accessed: 2023-04-24.

[5] Model of a Biological Neuron, Activation Functions, Neural Net architecture, Representational Power. `https://cs231n.github.io/neural-networks-1/`. Accessed: 2023-04-24.

[6] Rumelhart D. et al. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[7] Logistic Regression Cost Optimization Function. `https://pylessons.com/Logistic-Regression-part6`. Accessed: 2023-04-24.

[8] Stochastic Gradient Descent in Python. `https://medium.com/@ugurozcan108/stochastic-gradient-descent-in-python-26ade78e622b`. Accessed: 2023-04-24.

[9] Ruder S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[10] Tan C. et al. A Survey on Deep Transfer Learning. *arXiv preprint arXiv:1808.01974*, 2018.

[11] F. Zhuang et al. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[12] Szegedy C. et al. Going Deeper with Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[13] UC Merced Land Use Dataset. `http://weegee.vision.ucmerced.edu/datasets/landuse.html`. Accessed: 2023-04-25.