
PROIECT DE DIPLOMĂ

FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII
ȘI TEHNOLOGIA INFORMAȚIEI
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

FACULTATEA DE ELECTRONICĂ

TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

Specializarea:

Proiect de diplomă

Absolvent,



**Decan,
Prof.dr.ing. Ovidiu POP**

Președinte comisie,

2014

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ
TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

Departamentul

Titlul proiectului de diplomă:

Descrierea temei:

Locul de realizare:

Data emiterii temei:

Data predării temei:

Absolvent,



Conducător,



Director departament,

Declarație pe proprie răspundere privind autenticitatea lucrării de diplomă

Subsemnata/ul

legitimat cu seria nr. (conform copiei anexate prezentei declarații,
copie semnată și certificată "conform cu originalul"), autorul lucrării

elaborată în vederea susținerii examenului de finalizare a studiilor de licență,
la Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Programul de studiu
din cadrul Universității Tehnice din Cluj-Napoca
sesiunea a anului universitar 2024-2025

declar pe proprie răspundere că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate în textul lucrării și în bibliografie.

Declar că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.
Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de absolvire/licență/diplomă/disertație.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative,
respectiv anularea examenului de diplomă.

Sunt de acord ca, pe tot parcursul vieții, în cazul în care este necesar și se va dori verificarea autenticității lucrării mele să fiu identificat și verificat în baza datelor declarate de mine și conform copiei documentului de identitate menționat mai sus.

Data

Nume și prenume

Semnătura



Absolvent:

Conducător:

SINTEZA PROIECTULUI DE DIPLOMĂ

Avizul conducătorului

Conducător:



Absolvent,



1 Cuprins

1	Cuprins	1
2	Rezumat în limba engleză	3
2.1	State of the Art	3
2.2	Theoretical Fundamentals	4
2.3	Implementation	5
2.4	Experimental Results	6
3	Planificarea activității.....	11
4	Stadiul actual al domeniului.....	12
4.1	Descrierea temei abordate	12
4.2	Importanța temei abordate.....	12
4.3	Contribuții științifice și implementări existente	12
4.3.1	Sisteme de recunoaștere facială tradiționale	12
4.3.2	Metode moderne bazate pe CNN	13
4.4	Abordarea proprie	15
5	Fundamentare teoretică	16
5.1	Rețele convoluționale – concepte esențiale.....	16
5.2	Straturile convoluționale	17
5.2.1	Funcția de activare	17
5.2.2	Pooling	17
5.2.3	Stratul de aplativare.....	18
5.2.4	Straturi complet conectate.....	18
5.3	Arhitectura HRNet – structură și avantaj	18
5.4	Funcția de pierdere ArcFace – teorie și formulă	20
5.5	Normalizarea L2 – scop și efect.....	21
5.6	Politica de învățare – Optimizator, Decay, Warm-up	21
5.6.1	Algoritmul de optimizare Adam	22
5.6.2	Strategia de decădere a ratei de învățare	22
5.6.3	Încălzirea rețelei (Warm-up)	23
5.7	Validare și checkpointing – Rol și mecanisme	23
5.7.1	Validarea periodică	23
5.7.2	Checkpointing-ul automat.....	24
5.7.3	Vizualizarea evoluției cu TensorBoard.....	24
5.8	Evaluarea embedding-urilor și analiza erorilor	25

5.8.1	Embedding-urile faciale – scop și proprietăți	25
5.8.2	Curba ROC și AUC – evaluarea deciziilor binare	25
5.8.3	Histogramă a scorurilor de similaritate cosine.....	25
5.8.4	Analiza confuziilor – clasificare și erori frecvente	26
6	Implementarea soluției propuse	27
6.1	Mediul de dezvoltare și platforma hardware utilizată.....	27
6.2	Structura generală a programului – fluxul de execuție	28
6.3	Construirea modelului: HRNet și capetele de rețea	29
6.3.1	Arhitectura HRNet – descriere și implementare	29
6.3.2	Normalizarea L2 și stratul ArcFace – cod și corelare teoretică	31
6.3.3	Configurarea modelului pentru antrenare: Softmax și ArcFace	33
6.4	Funcția de pierdere ArcFace – explicație detaliată și cod.....	34
6.5	Procesul de antrenare – optimizare, validare, salvare	35
6.6	Evaluarea modelului – clasificare și analiză embedding	37
6.6.1	Evaluare clasică cu acuratețe și matrice de confuzie	37
6.6.2	Evaluare embedding cu similaritate cosine și ROC	39
7	Rezultate experimentale	41
7.1	Setul de date	41
7.1.1	Structurarea subsetului	41
7.1.2	Format și organizare.....	41
7.1.3	Convertirea în format TFRecord.....	42
7.1.4	Considerații practice.....	42
7.2	Metodologia experimentelor	43
7.2.1	Scenariile experimentale	43
7.2.2	Evaluare și metrici.....	43
7.3	Rezultate obținute.....	43
7.4	Interpretarea comparativă a rezultatelor.....	50
8	Concluzii	51
9	Bibliografie	52
10	Anexe	54

2 Rezumat în limba engleză

2.1 State of the Art

Facial recognition has evolved significantly over the past decades, becoming a central area within the field of computer vision, with widespread applications in security, biometric authentication, surveillance, and human-computer interaction. This thesis aims to develop a modern facial recognition system, built upon the HRNet architecture and trained with specialized loss functions such as ArcFace, to generate robust facial embeddings capable of reliably distinguishing identities in various real-world contexts.

The importance of this topic is highlighted by the growing impact of facial recognition technology in everyday life. From unlocking smartphones to controlling access in secure areas, such systems are already widely deployed. At the same time, their development raises challenges regarding personal data protection and privacy, requiring careful consideration of ethical and legal implications.

Technological advances have driven a shift from traditional methods, such as Eigenfaces, Fisherfaces, and LBPH, to modern solutions based on Convolutional Neural Networks (CNNs). These classic methods relied on linear projections and simple texture descriptors, which struggled with real-world variations in lighting, facial expression, and viewing angle. In contrast, CNNs are capable of automatically learning complex features directly from images, providing significantly improved robustness and performance.

However, even within deep learning approaches, the commonly used Softmax loss function has limitations in open-set recognition scenarios. This has led to increased interest in more advanced loss functions, such as ArcFace, which introduce an angular margin between classes to maximize their separation in the embedding space. As a result, models trained with ArcFace have achieved superior performance on benchmarks like LFW, MegaFace, and IJB-C.

Another key component is the HRNet architecture, which maintains high-resolution representations throughout the network and effectively fuses spatial information at multiple scales. This approach has proven especially effective in capturing fine-grained facial features, making HRNet an ideal choice for modern face recognition systems.

In this work, an open-source project inspired by InsightFace was adapted and extended, combining HRNet with both Softmax and ArcFace loss functions in a modular, trainable, and easily testable framework. The resulting architecture enables clear code organization, model checkpointing and restoration, and robust testing of facial embeddings using cosine distance. This solution offers a solid foundation for future development and deployment in real-world applications.

2.2 Theoretical Fundamentals

This chapter provides an overview of the essential concepts underlying the development of a facial recognition system. At its core are convolutional neural networks (CNNs), the HRNet architecture, and the ArcFace loss function – fundamental elements for understanding how a network can learn to distinguish facial identities in a discriminative latent space.

Convolutional networks are presented as a key tool in image processing, capable of automatically extracting relevant visual features directly from input data. Unlike traditional methods based on hand-crafted descriptors, CNNs construct a hierarchy of features, from edges and simple shapes to complex and meaningful structures. The defining components of these networks – convolutional layers, activation functions (ReLU), pooling, and fully connected layers are explained intuitively, highlighting the contribution of each in the information propagation flow.

A central activation example is the ReLU function, mathematically defined as:

$$f(x) = \max(0, x). \quad (1)$$

This introduces non-linearity, allowing the network to learn complex relationships and ignore negative activations. Together with pooling layers, which reduce the spatial dimensions of the data and help with generalization, and fully connected layers that finalize the decision, these mechanisms make up the basic structure of modern CNNs.

In more recent architectures, such as HRNet, dilated convolutions are often preferred instead of classical pooling to better preserve essential spatial details in the early stages of the network.

HRNet is regarded as a high-resolution network architecture, capable of preserving spatial fidelity across multiple parallel branches operating at different resolutions. Rather than downsampling the input aggressively, the model maintains several resolution streams in parallel, with information exchanged between them through upsampling and downsampling operations. The general architecture, comprising four successive processing stages, is illustrated in Figure 1.

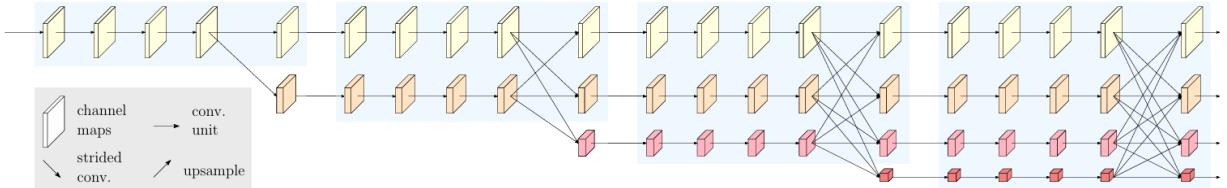


Figure 1. An example of a high-resolution network. Only the main body of the network is illustrated, excluding the stem part, which consists of two 3×3 convolutional layers with stride 2.

The network is divided into four stages.

This structure enables the simultaneous extraction of both fine-grained local features and high-level contextual information, which is particularly useful in computer vision tasks such as face detection, pose estimation, and facial identity recognition.

Another central concept is the ArcFace loss function, introduced as a more effective alternative to the classical Softmax. It imposes angular separation between classes in the embedding space, using a fixed angular margin to compact representations of faces from the same class and clearly separate those of different identities. The general formula of the function is:

$$L = -\frac{1}{N} \sum_{i=1}^N \log \left(\frac{e^{s \cdot \cos(\theta_{yi} + m)}}{e^{s \cdot \cos(\theta_{yi} + m)} + \sum_{j \neq yi} e^{s \cdot \cos(\theta_j)}} \right) \quad (2)$$

Where θ_{yi} is the angle between the feature vector and the center of the correct class, s is a scaling factor, and m is the artificial angular margin. This method has demonstrated superior ability to coherently structure the embedding space for open-set facial recognition.

Complementary to this, L2 normalization is considered a critical step to ensure that all embedding vectors are projected onto the unit hypersphere, allowing comparison based solely on angular distance. The normalization formula is:

$$\hat{x} = \frac{x}{\|x\|_2} = \frac{x}{\sqrt{\sum_{i=1}^n x_i^2}} \quad (3)$$

This transformation reduces the influence of scale and lighting variations in images, enabling robust similarity measurements between faces, especially when cosine-based metrics are used.

From an optimization perspective, several components contribute to effective training. The Adam optimizer is often employed, as it updates the network parameters based on a weighted average of gradients, according to the expression:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t + \epsilon}} \quad (4)$$

To further stabilize training and improve performance, a linear decay strategy for the learning rate can be applied, typically expressed as:

$$lr(t) = lr_{initial} \cdot \left(1 - \frac{t}{T}\right) \quad (5)$$

At the beginning of training, a warm-up strategy is commonly adopted, in which the learning rate increases gradually to prevent numerical instability during the early training epochs.

Validation is generally used to monitor the generalization capability of the network over time, while checkpointing mechanisms allow for the recovery or selection of model states based on performance criteria. These processes are often accompanied by the use of logging tools, which provide a clear visual representation of training dynamics and trends.

Finally, advanced evaluation methods are introduced for analyzing the quality of learned embeddings. These include the analysis of positive and negative image pairs, the calculation of cosine similarity scores, ROC curve generation, and AUC measurement – all reflecting the network's ability to coherently separate identities. Cosine similarity histograms and confusion matrices are typically used to visualize score distributions and identify common error patterns.

In summary, this chapter offers a rigorous and comprehensive theoretical foundation for understanding the architectural, algorithmic, and evaluation principles that underpin robust and high-performance facial recognition systems.

2.3 Implementation

The practical implementation of the facial recognition system was carried out on a personal laptop, using Python 3.6 and the TensorFlow 2.4 framework, within a conda virtual environment. Code editing was performed using Spyder, while the scripts were executed in the Anaconda Prompt. The system benefited from GPU support (NVIDIA RTX 3050 Ti), which allowed for efficient experiment execution, although training on the full MS1M-ArcFace dataset was not feasible due to memory constraints. As a result, a subset of 100 individuals was used, totaling over 7,000 images distributed for training, validation, and testing.

The software architecture is modular, with each file having a specific role: image loading, HRNet construction, application of the classification head (Softmax or ArcFace), loss computation, training execution, and evaluation. The complete workflow involves extracting embeddings from input images using HRNet, classifying or projecting them into the angular space (depending on the chosen head), computing the error, updating the weights using Adam, logging metrics with TensorBoard, and automatically saving the best-performing model checkpoints.

To extract facial features, HRNet v2 was used, a high-performance architecture that maintains high resolution throughout all stages of the network. It consists of four branches with different resolutions that communicate through upsampling and downsampling operations. The network output is a 512-dimensional embedding vector, obtained by concatenating the outputs and applying convolutions, global pooling, and dense layers.

Depending on the selected scenario, the embedding is either passed through a Dense layer with Softmax activation (for classical classification) or normalized using L2 and processed by the ArcLayer, a custom layer that computes the cosine between the embedding and the class vectors. In the case of ArcFace, the loss function modifies the score only for the correct class by adding an angular margin, enforcing a clearer geometric separation between identities.

Training is orchestrated by the TrainingSupervisor class, which manages the entire process: forward propagation, loss calculation, backpropagation, model saving, and logging in TensorBoard. The Adam optimizer is used, with parameters chosen for stability, along with a learning rate policy that includes both warm-up and linear decay. Periodic validation is integrated into every epoch, and the checkpoint with the best results is automatically retained.

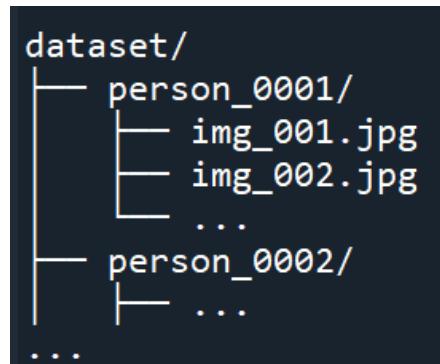
The model evaluation was performed in two stages. First, multi-class classification was applied directly to the embeddings, and the predicted scores were compared with the true labels. Accuracy was computed, and a complete confusion matrix was generated, along with a reduced version (highlighting the 20 most frequent confusions) in the form of a heatmap. Next, to analyze the embeddings, image pairs were formed, and cosine similarity was calculated. Based on these scores, an ROC curve was plotted, and the area under the curve (AUC) was used as a global metric. Cosine score histograms for positive and negative pairs provided additional insight into the discriminative capacity of the latent space.

All of these stages validate the completeness and robustness of the implemented solution, which offers a flexible system capable of learning high-quality embeddings and performing facial recognition in both closed-set (classification) and open-set (similarity-based verification) modes. Its modular and configurable implementation allows for future extensions toward larger datasets, alternative backbones, or specialized loss functions.

2.4 Experimental Results

To validate the effectiveness of the implemented facial recognition system, a series of rigorous experiments were conducted under controlled conditions. The dataset used was a representative subset extracted from MS1M-ArcFace, containing 7,208 images corresponding to 100 identities. This choice was dictated by the hardware limitations of the platform used (a personal laptop with a mid-range GPU) and the need to obtain fast yet meaningful results. The image distribution was balanced: approximately 69% for training, 14% for validation, and 16% for testing, following a stratified split. The final data format was converted to TFRecord for more efficient loading in TensorFlow. Additionally, a special binary file (test_pairs_lfw.bin) was generated for testing on positive and negative image pairs, based on the LFW benchmark model.

The structure of this subset and example images used are illustrated in Figure 2, providing a clear view of the dataset organization and the visual variability present in the set (lighting, angles, facial expressions):



Exemplu din dataset – Persoanele 0–4



Figure 2. Structure of the dataset subset and example images from the first five classes.

The evaluation included three training scenarios, each based on a different loss function configuration: the first scenario used the classical Softmax model, the second relied on the ArcFace model, which enforces angular separability, and the third was a hybrid model that combines both approaches – initially training with Softmax for 7 epochs, followed by 23 epochs with ArcFace. All three models were evaluated uniformly, under the same conditions, using the same test set and the best checkpoints saved based on validation performance.

The evaluation was carried out on two levels. First, direct classification was performed by comparing the predicted scores with the ground-truth labels. Accuracy was computed, and both a full confusion matrix and a reduced version (highlighting the 20 most frequent class confusions) were generated as heatmaps. Then, for embedding analysis, 1,000 image pairs (500 positive, 500 negative) were formed, and cosine similarity was calculated between them. Based on the resulting scores, ROC curves were plotted, and the area under the curve (AUC) was used as a global metric. Cosine score histograms for positive and negative pairs provided additional insight into the discriminative power of the latent space.

Table 1. Results obtained for each model on the test set.

Model	Accuracy (%)	Loss (loss)	ROC AUC
Softmax	89.20	8.5566	0.9920
ArcFace	89.46	8.6397	0.9930
Softmax + ArcFace	88.44	9.7116	0.9925

The ROC curves for the three models are shown in Figure 3, illustrating their ability to distinguish between positive and negative pairs:

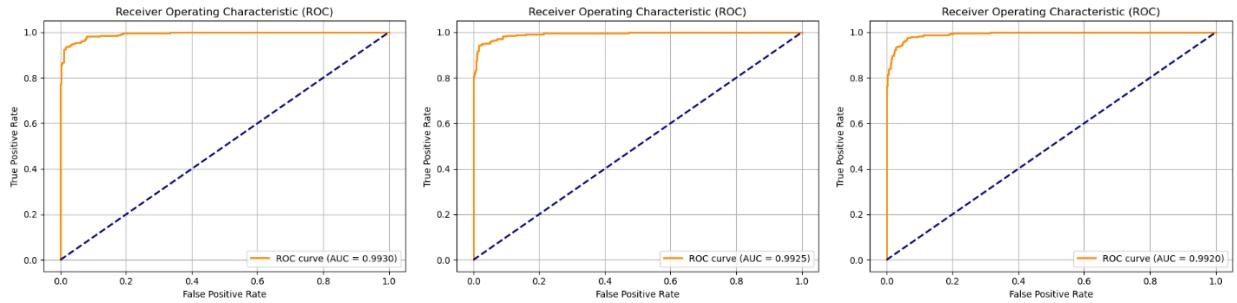


Figure 3. ROC curves for all three models (Softmax, ArcFace, Softmax + ArcFace).

To complement this analysis, Figure 4 presents the cosine similarity histograms, highlighting the distributions for positive pairs (same identity) and negative pairs (different identities):

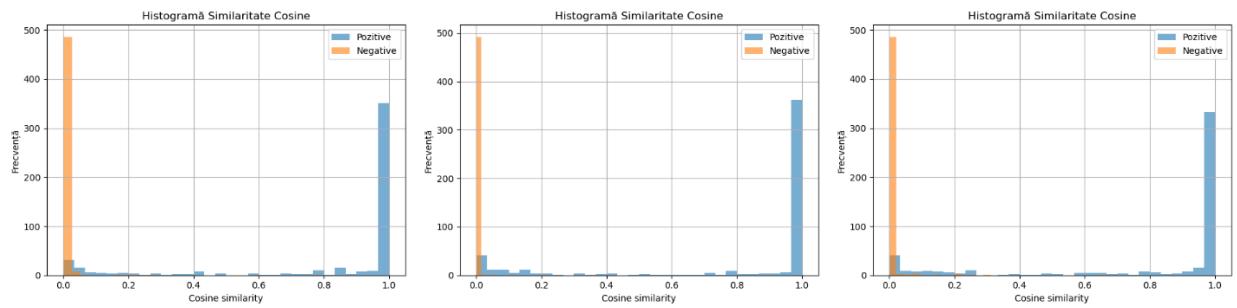


Figure 4. Cosine similarity histograms for the three models.

The results demonstrate high and relatively similar performance across the three methods, with excellent AUC values (above 0.99 in all cases). However, the differences become more apparent when analyzing their behavior in detail. To this end, three additional types of visualizations were introduced, providing a clearer perspective on the strengths and weaknesses of each model:

- Confusion heatmap: Figure 5 highlights the 20 most frequently confused class pairs, offering valuable insights into error patterns.
- Misclassifications: Figure 6 presents visual examples where the model made mistakes – the test image and the wrongly predicted image with the highest cosine similarity.
- Correct classifications: Figure 7 shows image pairs that were correctly classified with high cosine similarity, illustrating the robustness of the embeddings in favorable scenarios.

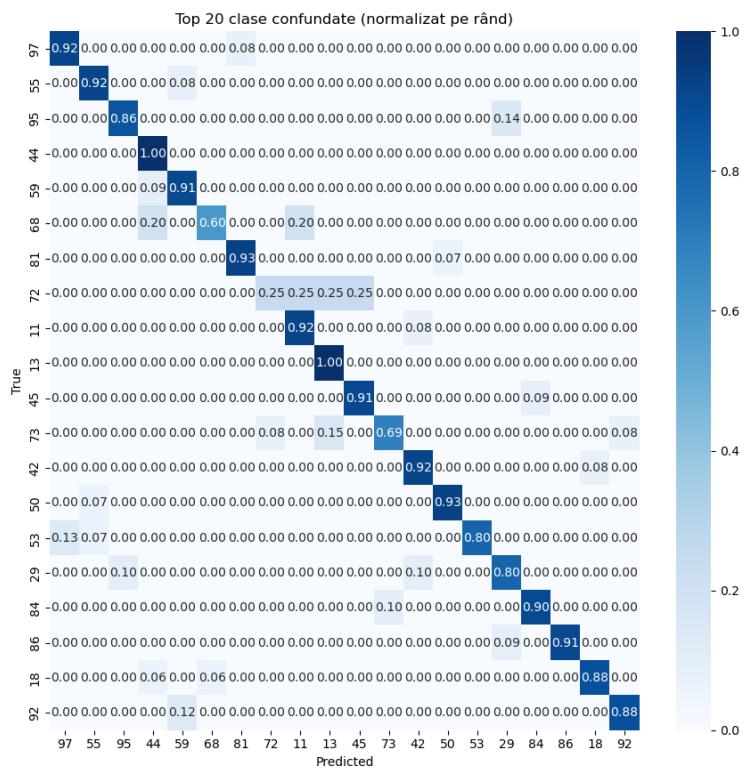


Figure 5. Heatmap of the most frequently confused class pairs.

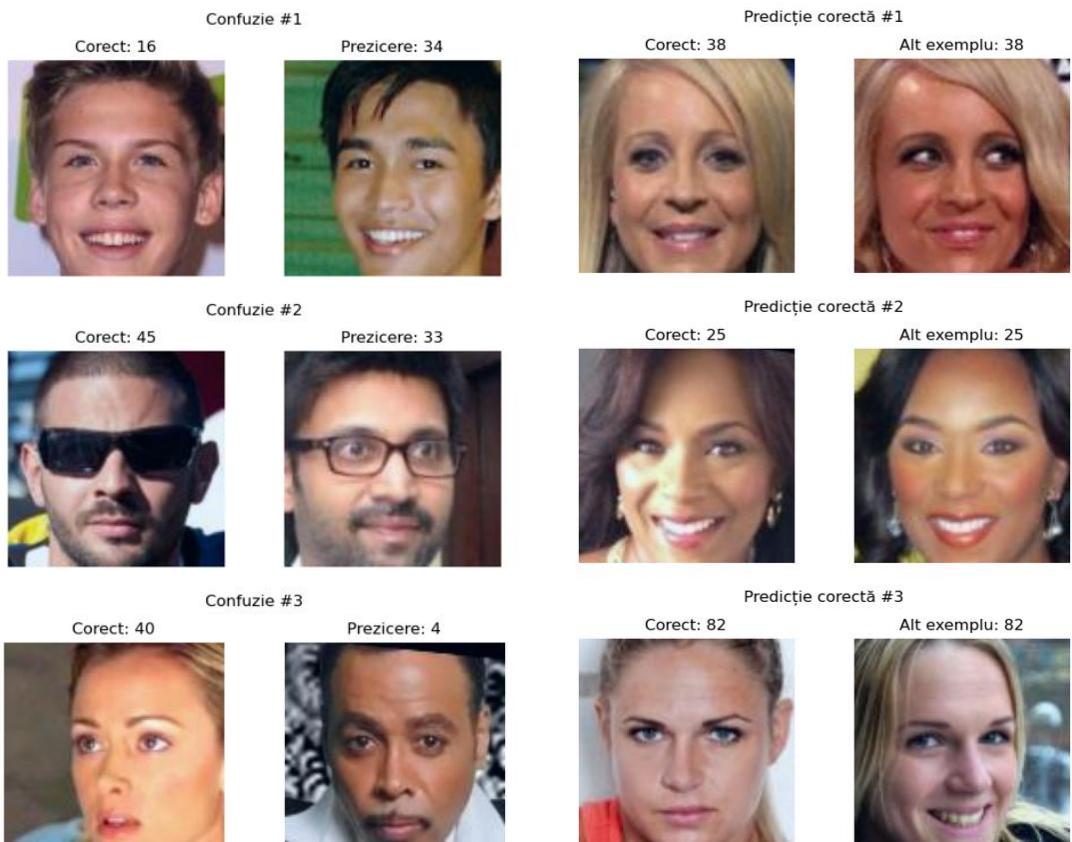


Figure 6. Visual examples of misclassified and correctly classified image pairs.

These visualizations contribute to a deeper understanding of how the networks behave under natural image variations (expressions, lighting, angles). They show that most errors are explainable and do not indicate fundamental flaws in the architecture. Additionally, they may suggest directions for improving the dataset or refining the face alignment process.

In conclusion, although the performance differences are small within the selected subset, the advantages of the ArcFace function become evident in more complex scenarios, where generalization and angular separation are essential. The comprehensive evaluation – both classical and embedding-based – confirms the robustness of the developed solution and validates the relevance of using the HRNet architecture along with a modular training strategy.

3 Planificarea activității

Nr. crt.	Activitate	Data de start	Nr. Zile	Data de sfârșit
1.	Documentare privind rețelele neuronale convoluționale (CNN) și recunoașterea facială	13.10.2024	20	01.11.2024
2.	Studierea arhitecturii HRNet și a funcției de pierdere ArcFace	02.11.2024	15	16.11.2024
3.	Configurarea mediului de lucru și testarea funcțională a pachetelor (TensorFlow, GPU etc.)	03.02.2025	10	12.02.2025
4.	Implementarea inițială a rețelei HRNet și adaptarea codului pentru antrenare	13.03.2025	20	01.04.2025
5.	Antrenarea modelelor în cele trei scenarii (Softmax, ArcFace, Softmax + ArcFace)	02.04.2025	15	16.04.2025
6.	Evaluarea modelelor și analiza rezultatelor (ROC, AUC, histograme, confuzii)	17.04.2025	20	06.05.2025
7.	Redactarea lucrării de diplomă și pregătirea pentru susținere	27.05.2025	35	30.06.2025

4 Stadiul actual al domeniului

4.1 Descrierea temei abordate

Recunoașterea facială a devenit o ramură esențială a viziunii artificiale, fiind utilizată în tot mai multe aplicații practice, precum autentificarea biometrică, supravegherea video, controlul accesului și identificarea persoanelor în rețelele sociale. De-a lungul timpului, abordările pentru recunoașterea facială au fost într-o continuă evoluție, trecând de la tehnici simple, bazate pe caracteristici vizuale ușor de extras, la sisteme complexe construite pe baza rețelelor neuronale profunde.

Această lucrare are ca obiectiv realizarea unui sistem de recunoaștere facială, utilizând o arhitectură avansată de rețea neuronală (HRNet) [Sun2019], în combinație cu două funcții de pierdere, una dintre ele fiind specializată în acest domeniu (ArcFace) [Deng2019]. Scopul este de a obține embedding-uri faciale robuste, care vor putea fi utilizate în clasificarea fețelor umane în aplicații practice.

4.2 Importanța temei abordate

Ne aflăm într-o perioadă în care metodele clasice de securitate nu mai fac față provocărilor actuale. Amenințările devin tot mai sofisticate și protecția datelor personale trebuie să țină pasul cu evoluția tehnologică. În acest context, biometria oferă soluții care combină eficiență cu un nivel ridicat de siguranță [Jain2011]. Dintre toate formele de autentificare biometrică, recunoașterea facială a devenit una dintre cele mai răspândite și accesibile metode.

O întâlnim deja în numeroase situații din viața de zi cu zi, cum ar fi:

- deblocarea telefonului prin recunoaștere facială (Apple Face ID, Android Face Unlock),
- verificarea identității în aplicații bancare sau instituții financiare,
- identificarea persoanelor în sisteme de supraveghere video,
- controlul accesului în aeroporturi, clădiri cu securitate sporită sau alte zone restricționate.

Totuși această temă poate crea un disconfort în cazul unor persoane deoarece în funcție de domeniu poate ridica semne de întrebare legate de dreptul la intimitate și viață privată. Din acest motiv dezvoltarea acestor sisteme trebuie să fie însotită de măsuri clare de protecție a datelor și de respectarea normelor etice și legale.

4.3 Contribuții științifice și implementări existente

4.3.1 Sisteme de recunoaștere facială tradiționale

Primele sisteme automate de recunoaștere facială se bazau pe identificarea unor trăsături geometrice de bază ale feței sau pe descriitorii simpli extrași direct din imaginea introdusă în sistem. Aceste metode pot fi grupate în două categorii: metode bazate pe caracteristici globale și metode bazate pe descriitorii locali. Câteva dintre principalele metode clasice sunt:

a) Metoda Eigenfaces (PCA)

Eigenfaces utilizează analiza componentelor principale (Principal Component Analysis – PCA) cu scopul reducerii dimensiunii imaginilor și extragerii direcțiilor principale de variație ale acestora [Turk1991]. Fiecare imagine este reprezentată sub forma unui vector, iar analiza componentelor principale (PCA) identifică un set de vectori ortogonali (denumiți „eigenfețe”),

folosiți pentru definirea unui spațiu subdimensional pentru reprezentarea fețelor.

Principiul de funcționare constă în:

- Calcularea imaginii medii a setului de antrenare;
- Determinarea vectorilor proprii asociați matricei de covarianță a datelor;
- Proiectarea fețelor în spațiul generat de acești vectori;
- Compararea noilor imagini față de proiecțiile deja existente, utilizând distanțe euclidiene.

Limitări: Această metodă este sensibilă la variații de iluminare, pozitie, orientare și expresie facială și nu poate modela relații complexe (non-liniare) între imagini.

b) Metoda Fisherfaces (LDA)

Metoda Fisherfaces reprezintă o extensie a analizei componentelor principale (PCA) și este bazată pe analiza discriminantă liniară (LDA – Linear Discriminant Analysis), pentru maximizarea separabilității dintre clase, adică între identitățile diferitelor persoane. Această metodă are ca scop construirea unui spațiu proiecțional în care variabilitatea între clase este maximizată, iar cea din interiorul clasei este minimizată. [Belhumeur1997].

Principiul de funcționare presupune:

- Proiectarea imaginilor feței într-un spațiu discriminativ, ghidat de etichetele fiecărei clase;
- Maximizarea raportului dintre dispersia inter-clasă și cea intra-clasă;
- Compararea proiecțiilor utilizând metriki de distanță.

Limitări: Precizia metodei scade dacă imaginile nu sunt bine aliniate sau dacă iluminarea variază semnificativ între probe.

c) Metoda Local Binary Patterns Histograms (LBPH)

LBPH este bazată pe extragerea de descriitori locali de textură ai feței, reprezentând una dintre cele mai populare tehnici tradiționale, datorită simplității și vitezei de execuție. Imaginea feței este împărțită în regiuni mici, uniforme, iar pentru fiecare pixel este calculat un cod binar comparând valoarea pixelului cu vecinii săi, rezultând un model caracteristic al texturii locale. [Ahonen2006]

Principiul de funcționare include:

- Aplicarea operatorului LBP pentru codificarea texturii locale;
- Construirea histogramelor de frecvență pentru fiecare regiune a imaginii;
- Concatenarea histogramelor într-un vector descriptor unitar;
- Compararea descriptorilor faciali utilizați printr-o metrică de distanță, precum distanța Chi-pătrat.

Limitări: Deși metoda este destul de robustă la variații moderate de iluminare, aceasta prezintă performanțe reduse în prezența zgromotului, variații de scară, rotație și expresii faciale accentuate, fiind eficientă în principal în condiții controlate și cu o eficiență limitată în scenarii din lumea reală.

4.3.2 Metode moderne bazate pe CNN

Odată cu dezvoltarea rețelelor neuronale conveoluționale (CNN), performanțele sistemelor de recunoaștere facială au fost îmbunătățite semnificativ [Krizhevsky2012]. Acestea au fost concepute pentru a prelucra de date cu structură spațială, cum ar fi imaginile, reușind să extragă în mod automat trăsăturile relevante folosindu-se de straturi conveoluționale, funcții de activare și operații de pooling [Krizhevsky2012]. Datorită arhitecturii lor ierarhice, CNN-urile pot învăța

reprezentări din ce în ce mai abstracte ale feței, de la forme simple la trăsături complexe.

Una dintre principalele funcții de pierdere folosite în rețelele neuronale este funcția Softmax. Aceasta a fost utilizată cu succes pentru diferite aplicații de clasificare, inclusiv în recunoașterea facială. Totuși, au fost observate limitări atunci când sistemele au fost aplicate în scenarii open-set, adică în situații în care apar persoane necunoscute față de datele de antrenare [Wang2018]. Chiar dacă funcția Softmax maximizează probabilitatea clasei corecte, nu garantează că embedding-urile fețelor aceleiași persoane sunt destul de apropiate și nici că embedding-urile persoanelor diferite sunt bine separate.

Lipsa de separabilitate între clase și de compactitate în interiorul clasei afectează semnificativ performanța în aplicații reale, unde identificarea corectă a fețelor noi este esențială. Mai mult, în contextul aplicațiilor critice, interpretabilitatea modelelor de învățare profundă devine esențială pentru a înțelege și valida deciziile automate, aspect subliniat și în cercetările recente privind transparența în ML (Machine Learning) [DoshiVelez2017].

Pentru a depăși aceste limitări, cercetările recente s-au concentrat pe dezvoltarea funcțiilor de pierdere specializate și a arhitecturilor avansate. Una dintre cele mai relevante progrese a fost introducerea funcțiilor de pierdere aditive cu margine unghiulară, iar dintre acestea, ArcFace s-a impus ca un standard în recunoașterea facială [Deng2019]. Această funcție ajustează procesul de învățare, adăugând o marjă suplimentară între clase, în spațiul normalizat (unitar), asigurând o abilitate mai bună de a distinge între identități și de a propria embedding-urile pentru aceeași persoană (clasă). Astfel, rețeaua devine mai robustă la variațiile de iluminare, expresii sau unghiuri care apar în imagini, oferind performanțe mai bune în scenariile open-set. Aceste performanțe au fost validate pe benchmark-uri standard precum LFW [Huang2008], MegaFace [KemelmacherShlizerman2016] și IJB-C.

Pe lângă funcțiile de pierdere, și arhitectura rețelei neuronale joacă un rol important în precizia recunoașterii faciale. În acest sens, HRNet (High-Resolution Network) se distinge datorită capacitații sale de a păstra și combina simultan multiple rezoluții pe tot parcursul propagării informației prin sistem [Sun2019], lucru care permite extragerea de detalii fine fără a pierde informație spațială. Această abordare este deosebit de eficientă în sarcini care implică localizarea precisă, precum detectia trăsăturilor faciale sau generarea de embedding-uri discriminative.

În practică, combinarea unui backbone robust precum HRNet cu o funcție de pierdere de tip ArcFace s-a dovedit a fi extrem de eficientă, lucru care a dus la rezultate de top în benchmark-uri de recunoaștere facială cunoscute, cum ar fi LFW (Labeled Faces in the Wild), MS-Celeb-1M sau IJB-C. Aceste rezultate semnificative au dus și la implementări industriale în domenii precum: sisteme de supraveghere video inteligente; acces biometric securizat; terminale de verificare a identității în aeroporturi sau instituții bancare; sisteme de autentificare facială pe dispozitive mobile și platforme online. Un cadru open-source de referință în acest sens este InsightFace [Guo2021].

Aceste evoluții arată clar cum recunoașterea facială a trecut de la metode generale de clasificare la soluții specializate, proiectate pentru a face față diversității și complexității fețelor umane în condiții reale.

O analiză sistematică recentă a metodelor CNN aplicate în recunoaștere facială confirmă aceste direcții, evidențiind importanța arhitecturii rețelei și a funcțiilor de pierdere adaptate [Barcic2023].

4.4 Abordarea proprie

Implementarea propusă în această lucrare se bazează pe proiectul open-source yinguobing/arcface, un sistem de recunoaștere facială inspirat din lucrarea de referință InsightFace [Guo2021]. Codul original a fost analizat și modificat pentru a corespunde cerințelor lucrării și pentru a îmbunătăți flexibilitatea și claritatea fluxului de antrenare.

Au fost introduse mecanisme de organizare și control al procesului de învățare, precum și suport pentru salvarea și restaurarea modelelor antrenate. De asemenea, arhitectura a fost modularizată pentru a permite selecția între capete de rețea diferite, HRNet fiind utilizată ca rețea principală pentru extragerea caracteristicilor faciale, în combinație cu funcțiile de pierdere Softmax și ArcFace. Seturile de date sunt organizate în directoare per clasă, iar evaluarea se face folosind distanțe cosine între embedding-uri normalize.

Această abordare asigură o bază solidă pentru dezvoltarea unui sistem modern de recunoaștere facială, care poate fi antrenat, testat și extins în funcție de cerințele aplicației vizate.

5 Fundamentare teoretică

Pentru a avea o înțelegere completă a tehnologiilor și deciziilor implicate în dezvoltarea sistemului propus, acest capitol prezintă concepțele teoretice fundamentale. Sunt explicate în detaliu noțiuni esențiale precum structura rețelelor neuronale conoluționale (CNN), arhitectura HRNet și funcția de pierdere ArcFace, alături de rolul normalizării L2 în generarea embedding-urilor faciale. În plus, sunt introduse principii importante privind politica de învățare (optimizare, ajustarea ratei de învățare și warm-up), validarea modelului și metodele de evaluare avansată a embedding-urilor (curbe ROC, histogramă de similaritate, matrice de confuzie). Toate aceste concepțe oferă baza teoretică necesară pentru înțelegerea implementării descrise în capitolul următor.

5.1 Rețele convoluționale – concepte esențiale

Rețelele neuronale convoluționale (CNN) formează o clasă importantă de arhitecturi utilizate în învățarea automată aplicată imaginilor. Acestea se inspiră din modul în care sistemul vizual uman prelucrează informațiile vizuale, imitând structura cortexului vizual. Spre deosebire de metodele tradiționale, unde extragerea trăsăturilor era realizată manual, CNN-urile pot învăța automat reprezentări relevante direct din datele de intrare, fiind extrem de eficiente în sarcini precum clasificarea și recunoașterea de obiecte [LeCun1998].

În prelucrarea imaginilor, rețelele convoluționale aplică în mod repetat filtre (numite și kernel-uri) pe regiuni locale din imagine. Aceste filtre detectează tipare vizuale esențiale, precum muchii, colțuri sau texturi. Pe măsură ce imaginea avansează prin rețea, straturile convoluționale extrag trăsături tot mai complexe, construind o reprezentare ierarhică și abstractă a conținutului vizual. Această reprezentare este apoi folosită în sarcini precum identificarea sau clasificarea obiectelor [Goodfellow2016].

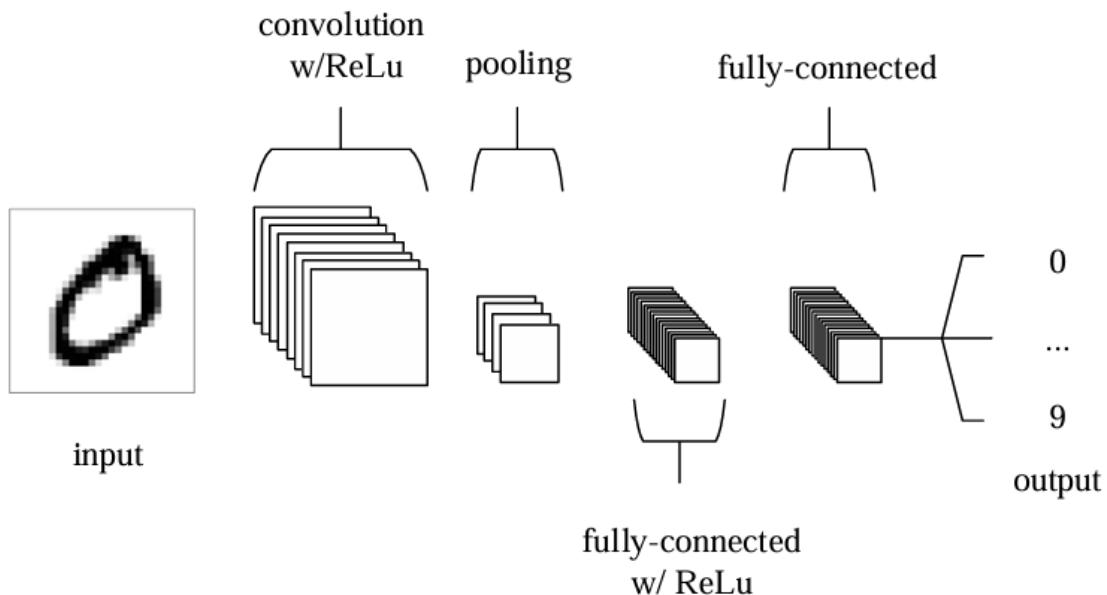


Figura 1. O arhitectură CNN simplă, compusă din doar cinci straturi [OShea2015].

Pentru a înțelege în profunzime cum funcționează rețelele convoluționale moderne, este esențial să examinăm principalele etape de procesare și componentele lor structurale.

5.2 Straturile convoluționale

Un prim element esențial este reprezentat de straturile convoluționale. Fiecare strat aplică un set de filtre care se deplasează peste imaginea de intrare sau peste ieșirea stratului anterior. Fiecare filtru învață să răspundă la o anumită caracteristică vizuală, cum ar fi marginile verticale sau orizontale. Acest mecanism local de învățare contribuie la reducerea semnificativă a numărului total de parametri și îmbunătățește capacitatea modelului de a generaliza [Goodfellow2016].

5.2.1 Funcția de activare

Funcția de activare cel mai des utilizată este ReLU (Rectified Linear Unit), aceasta fiind definată ca:

$$f(x) = \max(0, x). \quad (1)$$

Funcția introduce non-liniaritate în rețea, reprezentând un aspect esențial pentru ca modelul să poată învăța relații complexe între caracteristicile extrase [Nair2010]. În absența acestei funcții, rețeaua ar acționa ca un simplu model liniar, nereușind să capteze complexitatea și variațiile din datele de intrare.

Pentru a demonstra modul de funcționare al funcției ReLU, să considerăm următorul exemplu: presupunem că un strat intermediar din rețea returnează un vector de ieșire:

$$\mathbf{x} = S[-2.5, 0.0, 3.2].$$

Aplicând ReLU, rezultatul va fi:

$$\text{ReLU}(\mathbf{x}) = [0.0, 0.0, 3.2].$$

Se poate observa că valorile negative au fost eliminate, lucru care face ca rețeaua să răspundă doar la activări pozitive. Această simplă proprietate este esențială în a construi reprezentări ierarhice ale datelor, permitând rețelei să învețe din exemple variate și complexe.

5.2.2 Pooling

Pooling-ul, dintre care cel mai utilizat este max-pooling, are rolul de a reduce dimensiunea spațială a reprezentării, păstrând cele mai relevante informații. Această operație permite rețelei să extragă trăsături la rezoluții diferite, rezultând în reducerea riscului de overfitting și creșterea rezistenței modelului la variații minore [Goodfellow2016], cum ar fi deplasările sau deformările din imagini.

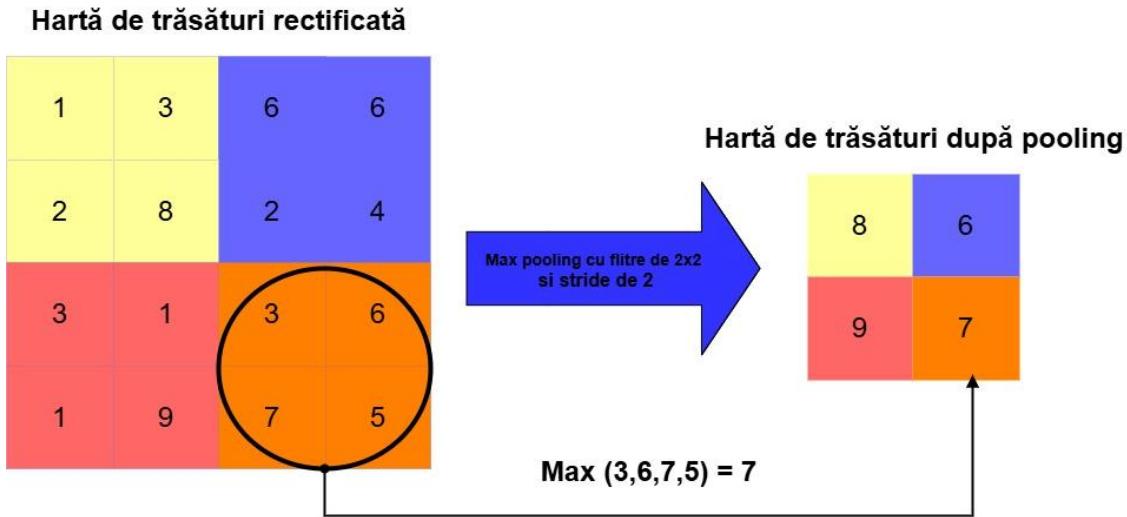


Figura 2. Exemplu de operație Max Pooling cu filtru 2×2 și pas (stride) 2 aplicată pe o hartă de trăsături.

În arhitecturi moderne precum HRNet, operațiile de pooling sunt adesea înlocuite cu convoluții cu pas mare (stride > 1), în special convoluții 3×3 cu pas 2, pentru a reduce rezoluția fără pierderi semnificative de informație. Această strategie permite păstrarea detaliilor spațiale importante, fiind mai adecvată în sarcini care necesită precizie ridicată, cum este recunoașterea facială.

5.2.3 Stratul de aplatizare

Stratul de aplatizare (flattening) are rolul de a transforma hărțile de trăsături multidimensionale (feature maps), rezultate din aplicarea filtrelor kernel, într-un vector unidimensional. Aceasă conversie este necesară pentru a putea conecta rezultatul rețelei la straturile complet conectate (fully connected), care funcționează ca un clasificator tradițional [Goodfellow2016].

5.2.4 Straturi complet conectate

Straturile complet conectate (Dense Layers) sunt cele în care fiecare neuron este conectat la toți neuronii stratului anterior. Acestea realizează funcția de decizie folosindu-se de trăsăturile extrase anterior.

În funcție de natura sarcinii sistemului neuronal, ultimul strat poate fi un softmax (pentru clasificare în clase discrete) sau un vector de embedding (pentru comparații pe baza distanței) [Goodfellow2016].

5.3 Arhitectura HRNet – structură și avantaj

High-Resolution Network (HRNet) [Sun2019] este o arhitectură avansată de rețea neuronală, proiectată pentru a păstra rezoluția înaltă a informației pe parcursul întregului proces de procesare. Spre deosebire de rețelele CNN tradiționale, care reduc treptat dimensiunea spațială a imaginilor prin pooling și convoluții cu pas mare, HRNet menține constant canalele de rezoluție înaltă și adaugă treptat ramuri suplimentare la rezoluții mai mici. Acest lucru permite păstrarea detaliilor fine și crește precizia în sarcini precum segmentarea semantică, detectia obiectelor sau recunoașterea facială.

Conceptul-cheie al HRNet constă în menținerea simultană a mai multor ramuri paralele, fiecare operând la o rezoluție diferită. În loc să reducă imaginea și apoi să o reconstruiască, cum fac rețelele de tip U-Net sau FPN, HRNet sincronizează aceste ramuri și permite schimb de informație între ele prin operații de upsampling și downsampling, urmate de agregare (fuziune).

Arhitectura completă este împărțită în patru stadii:

- Stadiul 1 aplică două convoluții 3×3 cu pas 2 (denumite „stem”), urmate de convoluții la rezoluție înaltă;
- Stadiul 2 introduce două ramuri;
- Stadiul 3 adaugă o a treia ramură;
- Stadiul 4 extinde structura la patru ramuri.

În fiecare stadiu, ramurile trec prin blocuri convoluționale (de tip Bottleneck sau Residual), iar între ele se aplică fuziunea prin Fusion Blocks. Aceste module folosesc:

- upsampling prin interpolare biliniară urmată de convoluții 1×1 , unde fiecare pixel nou este calculat prin media ponderată a celor patru vecini apropiati;
- downsampling prin convoluții 3×3 cu pas 2.

Astfel, fiecare ramură beneficiază de contextul celorlalte fără pierderi semnificative de informație.

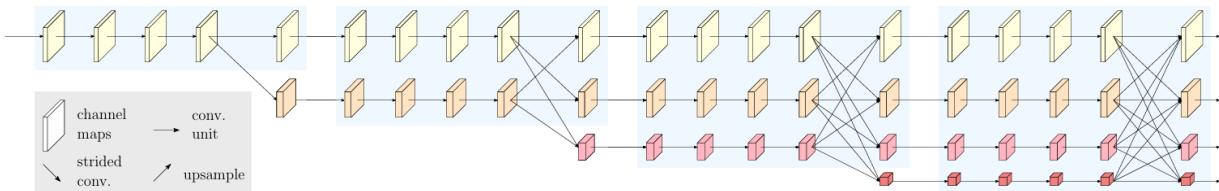


Figura 3. Un exemplu de rețea cu rezoluție înaltă. Este ilustrat doar corpul principal al rețelei, fără partea de început (stem), care constă în două straturi convoluționale 3×3 cu pas 2. Rețeaua este împărțită în patru stadii. Primul stadiu conține convoluții 1 [Sun2019].

Un avantaj clar față de alte backbone-uri (precum ResNet, MobileNet sau EfficientNet) este faptul că HRNet păstrează rezoluția ridicată pe tot parcursul rețelei. De exemplu, ResNet comprimă imaginea de până la 1/32 din dimensiunea sa inițială, ceea ce poate duce la pierderi importante de informație spațială, lucru nedorit în aplicații precum recunoașterea facială. HRNet, în schimb, menține detalii locale și context global, oferind embedding-uri mai precise.

Performanțele arhitecturii au fost demonstate în [Sun2019], unde HRNet a obținut:

- 75.1% mIoU pe setul Cityscapes pentru segmentare semantică (comparativ cu 72.0% pentru DeepLabv3+ și 71.4% pentru PSPNet);
- Locul I la COCO Keypoints Challenge 2019, în sarcina de estimare a poziției corpului uman (pose estimation).

În plus, HRNet este compatibilă cu tehnici moderne de optimizare:

- regularizare (penalizarea ponderilor mari, pentru reducerea overfitting-ului),
- prunare (eliminarea conexiunilor inutile pentru eficiență),
- cuantizare (reducerea preciziei numerice a ponderilor, utilă pentru dispozitive embedded sau aplicații mobile).

Această flexibilitate arhitecturală face din HRNet o alegere potrivită nu doar pentru cercetare, ci și pentru implementări industriale de la autentificare facială pe telefoane, la recunoaștere

facială locală în sisteme de supraveghere video de ultimă generație.

5.4 Funcția de pierdere ArcFace – teorie și formulă

În contextul recunoașterii faciale, precizia modelului nu depinde doar de arhitectura rețelei neuronale, ci și de tipul funcției de pierdere utilizate în timpul antrenării. În mod tradițional, clasificarea se face folosind funcția Softmax, care maximizează probabilitatea clasei corecte [Bridle, 1990]. Totuși, în aplicații precum recunoașterea facială, această abordare clasică nu este întotdeauna suficientă.

Pentru a înțelege această limitare, trebuie să ținem cont că Softmax are tendința de a evidenția doar scorul clasei corecte în raport cu celelalte, fără a asigura că reprezentările fețelor învățate sunt bine separate între persoane diferite și apropriate între imagini ale aceleiași persoane. Astfel, este posibil ca două imagini ale aceleiași persoane să fie plasate prea departe una de cealaltă în spațiul vectorial, iar imagini ale unor persoane diferite să ajungă prea aproape. Acest lucru poate reduce considerabil performanța în situații reale, unde sistemul trebuie să recunoască fețe noi, care nu au fost văzute în timpul antrenării.

Pentru a rezolva această problemă, cercetătorii au propus o funcție de pierdere specializată numită ArcFace [Deng2019]. Ideea din spatele acesteia este simplă, dar ingenioasă: în loc să lucreze cu produsul scalar dintre vectorul de intrare și vectorul de ponderi (ca în Softmax), ArcFace transformă acest produs într-un unghi, apoi adaugă o marjă fixă (angular margin) între clase. Astfel, două clase nu mai sunt separate doar prin distanță, ci prin unghi, într-un spațiu normalizat în care toți vectorii au aceeași lungime.

Pe scurt, dacă θ este unghiul dintre dintre vectorul de caracteristici al unui exemplu și vectorul de ponderi al clasei corecte, ArcFace înlocuiește $\cos(\theta)$ cu $\cos(\theta+m)$, unde m este o marjă unghiulară (de obicei 0.5). Această ajustare obligă rețeaua să „muncească” mai mult pentru a aduce exemplele din aceeași clasă mai aproape unii de alții și să separe mai clar clasele diferite.

Matematic, funcția devine:

$$L = -\frac{1}{N} \sum_{i=1}^N \log \left(\frac{e^{s \cdot \cos(\theta_{y_i} + m)}}{e^{s \cdot \cos(\theta_{y_i} + m)} + \sum_{j \neq y_i} e^{s \cdot \cos(\theta_j)}} \right) \quad (2)$$

unde:

- N este numărul de exemple din batch,
- s este un factor de scalare (ex: 64),
- m este marja unghiulară (ex: 0.5),
- θ_{y_i} este unghiul dintre vectorul de intrare și vectorul clasei corecte,
- θ_j sunt unghiurile pentru celelalte clase.

Această formulă pare complexă, dar esența este că în loc să comparăm doar distanțe brute între vectori, comparăm unghiuri, ceea ce face sistemul mult mai robust la variații de scară și lumină.

Experimentele din literatura de specialitate au arătat că ArcFace oferă rezultate de top în benchmark-uri precum LFW, MegaFace și IJB-C, depășind alte metode precum SphereFace sau CosFace. În plus, este relativ ușor de implementat și se comportă stabil în timpul antrenării.

Așadar, ArcFace nu este doar o funcție de pierdere cu o formulare matematică mai riguroasă, ci și o soluție practică la o problemă concretă din recunoașterea facială: obținerea de embedding-uri discriminative, care pot fi comparate eficient în timpul inferenței.

5.5 Normalizarea L2 – scop și efect

În contextul rețelelor neuronale utilizate pentru recunoaștere facială, o etapă esențială în procesarea embedding-urilor este normalizarea L2. Această tehnică transformă fiecare vector de caracteristici astfel încât lungimea sa (norma L2) să fie egală cu 1. Practic, vectorul este „proiectat” pe suprafața unei sfere unitare.

Această operație este crucială pentru a obține embedding-uri care pot fi comparate corect între ele, în special când se folosește distanța cosine ca măsură de similaritate.

Pentru un vector de caracteristici $x \in \mathbb{R}^n$, normalizarea L2 se face astfel:

$$\hat{x} = \frac{x}{\|x\|_2} = \frac{x}{\sqrt{\sum_{i=1}^n x_i^2}} \quad (3)$$

Explicația fiecărui termen:

- x : vectorul original, format din n componente, rezultat dintr-o rețea neuronală (embedding).
- $\|x\|_2$: norma L2 a vectorului, adică lungimea sa în spațiul euclidian.
- $\sum_{i=1}^n x_i^2$: suma pătratelor fiecărei componente a vectorului.
- \hat{x} : vectorul rezultat după normalizare — are aceeași direcție ca x , dar lungimea fixă de 1.

După ce vectorul de caracteristici (embedding-ul) este generat de rețea, este important ca el să poată fi comparat în mod corect cu alte embedding-uri, indiferent de condițiile în care au fost obținute. De exemplu, două imagini ale aceleiași persoane, una realizată în lumină naturală și cealaltă în condiții de iluminare slabă, pot produce vectori de dimensiuni (magnitudini) diferite, chiar dacă direcția lor în spațiul caracteristic este foarte apropiată. În astfel de situații, normalizarea L2 devine esențială: ea elimină influența magnitudinii și păstrează doar direcția, care este relevantă pentru recunoaștere.

Un alt motiv pentru aplicarea acestei tehnici este compatibilitatea cu funcții de pierdere avansate, precum ArcFace, care evaluează unghiul dintre vectorul de intrare și vectorul de ponderi al unei clase. Acest unghi poate fi interpretat corect doar dacă ambele vectori sunt normalizați. Altfel, distanțele și deciziile în spațiul embedding devin instabile și greu de interpretat [Ranjan2017].

Pe lângă toate acestea, normalizarea L2 proiectează toate embedding-urile pe suprafața unei sfere unitare, ceea ce face ca distanțele și relațiile dintre puncte să fie uniforme și ușor de analizat. În acest cadru geometric, metodele de clasificare devin mai clare, iar operații precum clustering-ul sau analiza de similaritate devin mai fiabile. Rețeaua învăță astfel să plaseze fețele asemănătoare aproape unele de altele și să le separe pe cele diferite, fără a fi influențată de factori perturbatori precum contrastul sau mărimea imaginii.

În concluzie, normalizarea L2 este mai mult decât un pas matematic, este o componentă crucială în obținerea de embedding-uri coerente, robuste și comparabile, care pot fi folosite cu încredere în orice aplicație modernă de recunoaștere facială.

5.6 Politica de învățare – Optimizator, Decay, Warm-up

Antrenarea eficientă a unei rețele neuronale profunde presupune mai mult decât alegerea unei arhitecturi potrivite. Un rol esențial îl joacă politica de învățare, adică modul în care parametrii rețelei sunt actualizați în funcție de eroarea obținută în timpul procesului de optimizare. Această politică este compusă, în general, dintr-un algoritm de optimizare (care decide cum se face

actualizarea), o strategie de ajustare a ratei de învățare (care controlează cât de mult se modifică parametrii la fiecare pas) și, uneori, o fază de încălzire (warm-up) care facilitează un început mai stabil al antrenării.

5.6.1 Algoritmul de optimizare Adam

Unul dintre cei mai utilizați algoritmi de optimizare în rețelele neuronale moderne este Adam (Adaptive Moment Estimation), propus de Kingma și Ba în 2015 [Kingma2015]. Adam îmbină avantajele algoritmilor AdaGrad și RMSProp, reușind să adapteze în mod automat rata de învățare pentru fiecare parametru al modelului.

Spre deosebire de algoritmul clasic SGD (Stochastic Gradient Descent), care folosește o rată de învățare fixă, Adam ajustează această rată folosind două momente statistice:

- media gradientului (numită „momentul de ordinul I”),
- media pătratelor gradientului (numită „momentul de ordinul II”).

Această adaptare are ca efect o convergență mai rapidă și mai stabilă, mai ales în cazul unor rețele adânci sau al unor date zgomotoase.

Formula de actualizare folosită de Adam este următoarea:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t + \epsilon}} \quad (4)$$

- θ_t sunt parametrii modelului la pasul t
- α este rata de învățare (learning rate),
- \widehat{m}_t este media gradientului (corectată),
- \widehat{v}_t este varianța gradientului (corectată),
- ϵ este un mic termen pentru stabilitate numerică.

Adam este în general preferat în aplicații practice deoarece oferă rezultate competitive „din prima”, fără necesitatea unei calibrări foarte fine a hiperparametrilor, ceea ce îl face ideal pentru proiecte experimentale sau prototipuri.

5.6.2 Strategia de decădere a ratei de învățare

Pe parcursul antrenării, este important ca rata de învățare să nu rămână constantă. În primele epoci, când modelul este încă departe de un minimum al funcției de pierdere, o rată de învățare mare poate accelera progresul. În schimb, spre final, pașii mari pot face ca modelul să oscileze în jurul unei soluții bune fără a se stabiliza. De aceea, în multe cazuri, se aplică o strategie de decădere.

Una dintre cele mai simple și eficiente metode este decăderea liniară a ratei de învățare. Aceasta presupune ca valoarea ratei să scadă treptat, într-un mod proporțional cu progresul în antrenare. Formula este:

$$lr(t) = lr_{initial} \cdot \left(1 - \frac{t}{T}\right) \quad (5)$$

unde:

- $lr(t)$ este rata de învățare la pasul t
- $lr_{initial}$ este valoarea inițială setată,
- T este numărul total de pași de antrenare.

Această metodă este apreciată pentru simplitatea sa și pentru faptul că permite o tranziție lină între faza de învățare agresivă (la început) și faza de rafinare fină a modelului (spre final). Alte metode mai complexe includ decăderea exponențială, scăderea la platou (reduce on plateau) sau politicile ciclice [Smith2017].

5.6.3 Încălzirea rețelei (Warm-up)

Warm-up-ul este o tehnică utilizată frecvent în antrenarea rețelelor profunde, în special în combinație cu optimizatori adaptați precum Adam sau când se folosesc batch-uri mari. Ideea de bază este că, în primele iterații, gradiențele pot fi instabile, iar pașii de actualizare prea mari pot „arunca” rețeaua într-o regiune neproductivă a spațiului parametrilor.

Pentru a evita acest fenomen, rata de învățare este crescută progresiv, de la o valoare foarte mică (chiar aproape de zero) până la valoarea dorită. Acest lucru se întâmplă de obicei în primele câteva epoci. După perioada de încălzire, rata de învățare urmează apoi o strategie de decădere (cum ar fi cea liniară).

Avantajul acestei metode este că oferă o tranziție stabilă în faza inițială a învățării și reduce riscul de divergență. Ea a fost demonstrată ca fiind eficientă în lucrări importante din domeniu, în special în antrenarea rețelelor adânci pentru clasificare de imagini la scară mare [Goyal2017].

5.7 Validare și checkpointing – Rol și mecanisme

Pe parcursul antrenării unui model de învățare profundă, este important ca performanța rețelei să fie monitorizată în mod constant, iar progresul să fie salvat într-un mod sigur și reproductibil. În general, aceste obiective se realizează prin două mecanisme complementare:

- validarea periodică – pentru estimarea capacitatei de generalizare;
- checkpointing-ul – pentru salvarea automată a versiunilor optime ale modelului.

Aceste tehnici nu doar că oferă un control mai bun asupra procesului de antrenare, dar permit și reluarea experimentului în caz de întreruperi, precum și alegerea celei mai bune versiuni a modelului pentru utilizare ulterioară. Detaliile legate de implementarea acestor mecanisme sunt prezentate în secțiunile următoare.

5.7.1 Validarea periodică

Validarea constă în evaluarea modelului pe un set de date diferit de cel folosit pentru antrenare. Acest set de validare conține exemple „nevăzute” de către rețea, dar care provin din aceeași distribuție ca datele de antrenare. Scopul este să estimăm capacitatea de generalizare a modelului – adică performanța sa pe date noi.

La finalul fiecărei epoci, se pot calcula metrii precum:

- acuratețea (accuracy) – proporția de exemple clasificate corect;
- valoarea funcției de pierdere – un indicator numeric al erorii medii pe setul de validare.

Valorile acestor metrii oferă indicii importante despre:

- supraînvățare (overfitting) – dacă performanța pe datele de antrenare continuă să crească, dar cea pe validare stagniază sau scade;
- subantrenare (underfitting) – dacă performanța e slabă pe ambele seturi.

Conform studiilor de referință [Goodfellow2016], validarea este recomandată la intervale regulate (de exemplu, la fiecare epocă) pentru a ghida corect procesul de antrenare și ajustarea hiperparametrilor.

5.7.2 Checkpointing-ul automat

Checkpointing-ul este procesul de salvare periodică a stării modelului, inclusiv:

- valorile parametrilor rețelei neuronale,
- starea optimizatorului,
- eventual și metrica de performanță la acel moment.

Motivația checkpointing-ului este dublă:

- Securizarea progresului – dacă procesul de antrenare este întrerupt, poate fi reluat de la ultimul punct salvat, fără a pierde epoci întregi.
- Păstrarea celei mai bune versiuni – dacă un model atinge o performanță maximă pe validare la epoca 14, iar apoi începe să degradeze, checkpoint-ul aferent epocii 14 poate fi restaurat pentru inferență.

În mod obișnuit, se salvează automat:

- fie modelul cu cea mai mică pierdere pe validare,
- fie modelul cu cea mai mare acuratețe,
- fie modele la intervale regulate (ex: la fiecare 5 epoci), pentru comparații ulterioare.

Această strategie este standard în multe cadre moderne de antrenare și este recomandată în special când resursele de calcul sunt limitate sau când experimentul are o durată mare [Bengio2012].

5.7.3 Vizualizarea evoluției cu TensorBoard

Pe lângă validarea numerică, este extrem de util ca antrenarea modelului să fie însoțită de o vizualizare grafică a metricilor, pentru a putea identifica rapid tendințe sau probleme. O unealtă standard pentru această sarcină este TensorBoard, dezvoltată de Google și integrată în cadrul TensorFlow.

Prin logarea valorilor pentru pierdere, acuratețe și rata de învățare, TensorBoard permite:

- urmărirea în timp real a curbelor de învățare;
- compararea diferitelor rulari ale modelului;
- identificarea timpurie a overfitting-ului (diferență mare între antrenare și validare).

Acest tip de analiză vizuală este recomandat de majoritatea lucrărilor moderne din domeniul deep learning, deoarece oferă intuiții rapide și ușor de interpretat [Abadi2016].

5.8 Evaluarea embedding-urilor și analiza erorilor

După antrenarea unei rețele neuronale pentru recunoaștere facială, este esențial să evaluăm nu doar acuratețea generală, ci și calitatea reprezentărilor generate. În acest sens, embedding-urile faciale obținute de model trebuie testate pentru a verifica dacă acestea sunt suficient de discriminative (separă bine identitățile diferite) și coerente (grupează apropiat imaginile aceleiași persoane). În plus, pentru modelele care fac clasificare multi-clasă, este importantă și analiza tipelor de eroare prin instrumente precum matricea de confuzie și reprezentări grafice.

5.8.1 Embedding-urile faciale – scop și proprietăți

Embedding-urile faciale sunt vectori numerici de dimensiune fixă (de exemplu, 512 dimensiuni), care reprezintă trăsăturile esențiale ale unei fețe într-un spațiu latent. Un model de recunoaștere facială reușit va învăța să plaseze imaginile aceleiași persoane cât mai aproape unele de altele și să mențină o distanță mare între persoane diferite [Wang2018].

Evaluarea acestor reprezentări se bazează adesea pe analiza similarității între perechi de embedding-uri:

- Perechi pozitive: două imagini ale aceleiași persoane → scor mare de similaritate.
- Perechi negative: două imagini ale unor persoane diferite → scor mic de similaritate.

Astfel, performanța rețelei nu mai este măsurată printr-o simplă clasificare, ci prin capacitatea de a distinge între aceste două tipuri de perechi [Deng2019].

5.8.2 Curba ROC și AUC – evaluarea deciziilor binare

Pentru evaluarea spațiului embedding, una dintre cele mai utilizate metode este construirea curbei ROC (Receiver Operating Characteristic). Această curbă descrie compromisurile între: Rata de pozitivi adevărați (True Positive Rate – TPR) și rata de pozitivi falși (False Positive Rate – FPR), în funcție de diferite praguri de decizie aplicate asupra scorului de similaritate (ex: cosine similarity).

O rețea performantă va genera o curbă ROC care se apropie de colțul stânga-sus, ceea ce înseamnă o detecție bună a perechilor pozitive și o rată mică de confuzii. Pentru cuantificarea globală a performanței, se calculează aria sub curbă (AUC – Area Under Curve). Aceasta variază între 0.5 (aleator) și 1.0 (separare perfectă), iar valori peste 0.95 sunt considerate excelente în contextul verificării faciale [Hanley1982].

ROC și AUC sunt utilizate frecvent în aplicații biometrice, unde sistemul trebuie să decidă dacă două fețe aparțin aceleiași persoane, fără a face parte dintr-un set predefinit de clase (scenarii open-set).

5.8.3 Histogramă a scorurilor de similaritate cosine

Pe lângă curba ROC, o metodă intuitivă de analiză a embedding-urilor constă în construirea unei histograme a scorurilor cosine, separate pe:

- perechi pozitive (care ar trebui să aibă scoruri apropiate de +1),
- perechi negative (care ar trebui să aibă scoruri mai mici, în jurul valorii 0 sau sub).

Această histogramă permite observarea gradului de suprapunere între cele două distribuții. Dacă distribuțiile sunt bine separate, înseamnă că embedding-urile sunt de calitate și permit alegerea unui prag de decizie robust. În schimb, o suprapunere semnificativă indică o rețea care nu a învățat să diferențieze clar identitățile, fiind predispusă la confuzii [Ranjan2017].

Histogramele de similaritate sunt utile nu doar pentru evaluare, ci și pentru stabilirea pragurilor numerice în aplicații practice (ex: accept/reject într-un sistem biometric) [Grother2019].

5.8.4 Analiza confuziilor – clasificare și erori frecvente

În cazul în care modelul este antrenat pentru clasificare directă (ex: folosind funcția Softmax sau ArcFace ca strat de decizie), o metodă importantă de evaluare o reprezintă matricea de confuzie. Aceasta este un tabel în care:

- rândurile corespund etichetelor reale,
- coloanele corespund predicțiilor modelului,
- elementele din interior indică numărul de imagini clasificate într-un anumit mod.

Pe diagonală se regăsesc predicțiile corecte, iar valorile din afara acesteia semnalează erorile. O matrice de confuzie „bună” are majoritatea valorilor concentrate pe diagonală. În schimb, o dispersie mare sugerează confuzii între clase.

Pentru o analiză mai detaliată, este utilă generarea unui heatmap al confuziilor, care evidențiază vizual cele mai frecvent confundate clase. Acest tip de analiză ajută la descoperirea perechilor de identități dificil de diferențiat – de exemplu, persoane cu trăsături faciale similare, iluminare slabă sau imagini nealiniate [Zhou2021].

De asemenea, prezentarea unor exemple vizuale ale greșelilor de clasificare (imaginile de test + o imagine a clasei prezise greșit) oferă intuiții suplimentare despre comportamentul modelului. Această metodă este folosită frecvent în procesul de „debugging vizual” al rețelelor neurale [DoshiVelez2017].

6 Implementarea soluției propuse

6.1 Mediul de dezvoltare și platforma hardware utilizată

Implementarea proiectului a fost realizată local, pe un laptop personal, fără a utiliza servicii de procesare în cloud. Mediul principal de dezvoltare a fost Anaconda, în cadrul căruia am folosit editorul Spyder pentru scrierea și organizarea codului, iar rularea efectivă a scripturilor s-a realizat din linia de comandă, în Anaconda Prompt, în cadrul unui mediu virtual dedicat denumit arcface_py36.

Proiectul a fost dezvoltat în limbajul Python, versiunea 3.6.13, utilizând framework-ul TensorFlow 2.4.0 și biblioteci auxiliare precum NumPy, Matplotlib, Seaborn, Scikit-learn și MTCNN. Mediul virtual a fost creat și gestionat cu ajutorul conda, iar toate pachetele necesare au fost instalate din canalele oficiale și pypi, conform cerințelor versiunii TensorFlow utilizate.

Echipamentul hardware folosit pentru antrenare este următorul:

- Procesor: Intel Core i5-11400H, 11th Gen @ 2.70 GHz
- Memorie RAM: 16 GB DDR4
- Placă video: NVIDIA GeForce RTX 3050 Ti Laptop GPU
- Sistem de operare: Windows 11, 64-bit

Utilizarea plăcii video a fost posibilă datorită instalării versiunilor compatibile ale CUDA (11.2.2) și cuDNN (8.1.0.77), pachete incluse în mediul conda configurat.

Pentru monitorizarea antrenării și logarea valorilor relevante (precum loss, acuratețe, rată de învățare etc.), am folosit TensorBoard, activat automat de clasa TrainingSupervisor. La fiecare epocă, sunt salvate evenimente TensorFlow în directorul logs/, care pot fi accesate cu comanda:

```
tensorboard --logdir logs/hrnetv2_softmax+arcface
```

Deoarece sistemul nu permite antrenarea completă a datasetului MS1M-ArcFace (85.000 identități, 5.8 milioane imagini), am extras un subset reprezentativ de 100 de persoane (7.208 imagini), folosit pentru toate experimentele. De asemenea, a fost necesară reducerea mărimii batch-ului la 8 imagini, pentru a evita erorile de memorie și pentru a menține procesul de antrenare stabil.

Antrenările au fost realizate în trei scenarii separate:

- Model 1 – doar Softmax (30 epoci)
- Model 2 – doar ArcFace (30 epoci)
- Model 3 – Softmax + ArcFace în două faze (7 + 23 epoci)

Pentru exportul modelelor antrenate s-a folosit argumentul --export_only=True, iar restaurarea ponderilor s-a realizat cu --restore_weights_only=True. Modelele salvate au fost ulterior utilizate în scriptul evaluate.py pentru testare și analiză detaliată a performanței.

6.2 Structura generală a programului – fluxul de execuție

Sistemul propus de recunoaștere facială este organizat modular, fiecare etapă fiind implementată într-un fișier separat, cu o responsabilitate bine definită. Această structură facilitează atât înțelegerea logicii generale, cât și extinderea ulterioară a funcționalităților.

Fluxul de rulare al aplicației urmează pașii logici necesari într-un sistem de recunoaștere facială modern, pornind de la imaginea brută și terminând cu predicția clasei sau analiza embedding-urilor. Pașii principali sunt:

1. Încărcarea imaginilor din fișiere .tfrecord, realizată prin funcția build_dataset() definită în dataset.py. Aici sunt decodeate imaginile, normalizate și etichetate sub formă one-hot.
2. Propagarea imaginilor prin rețea, utilizând modelul HRNet construit în network.py. Aceasta produce, pentru fiecare imagine, un vector de embedding de dimensiune 512.
3. Tratarea embedding-ului în funcție de modul de antrenare:
 - Dacă este selectat modul clasic, embedding-ul este trimis într-un strat Dense urmat de Softmax.
 - Dacă este selectat ArcFace, embedding-ul este normalizat L2 și trecut printr-un strat ArcLayer care proiectează pe spațiul claselor.
4. Calculul pierderii, realizat prin funcția loss_fun, care este aleasă în train.py în funcție de argumentul --softmax. Pierdere este aplicată ulterior în TrainingSupervisor, unde este combinată cu regularizările rețelei.
5. Derivarea gradientului și actualizarea ponderilor, făcute în cadrul metodei _train_step() din training_supervisor.py. Aici se utilizează optimizatorul Adam și se aplică apply_gradients() pe toate ponderile rețelei.
6. Logarea performanței în TensorBoard și salvarea checkpoint-urilor, efectuate automat la intervale configurabile, pentru a păstra versiunile cele mai performante ale modelului.
7. Evaluarea finală a modelului, realizată prin scriptul evaluate.py, care oferă atât analiză clasică de clasificare (cu acuratețe și matrice de confuzie), cât și evaluare bazată pe embedding (distanță cosine, ROC, histogramă).

Acești pași sunt organizați vizual în schema logică de mai jos, care prezintă succesiunea operațiilor în sistemul implementat:

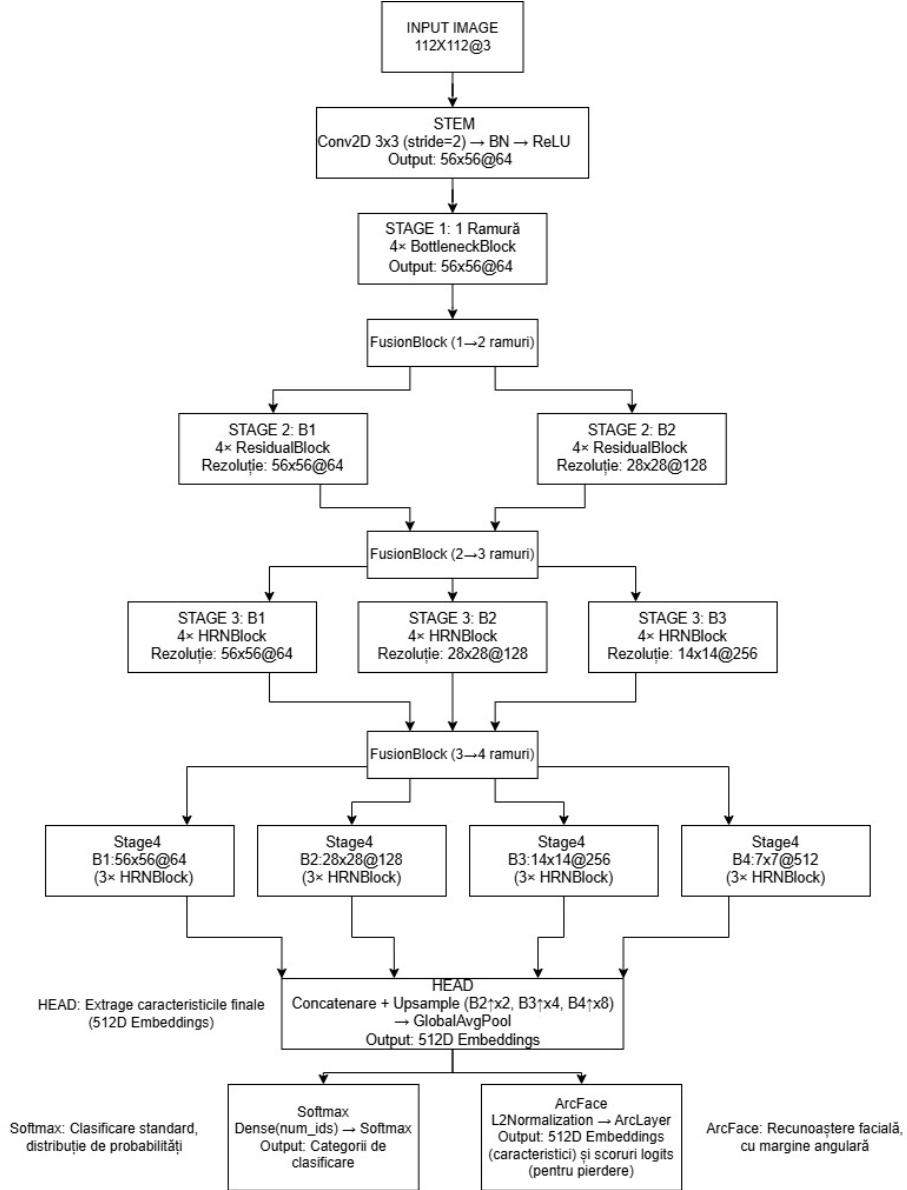


Figura 4. Arhitectura completă a modelului propus, cu HRNet și cap de rețea Softmax / ArcFace.

Această diagramă ilustrează parcursul complet al unei imagini prin sistem: încărcare, trecere prin rețea, extragere de trăsături (embedding), clasificare sau măsurare a similarității, și afișarea rezultatului. Structura modulară a codului permite înlocuirea ușoară a unor componente (ex: backbone-ul rețelei sau funcția de pierdere) fără a afecta restul implementării.

6.3 Construirea modelului: HRNet și capetele de rețea

6.3.1 Arhitectura HRNet – descriere și implementare

Pentru extragerea embedding-urilor faciale am folosit arhitectura HRNet (High-Resolution Network), recunoscută pentru capacitatea sa de a păstra detaliile spațiale prin întreaga rețea. Acest model a fost utilizat ca backbone pentru rețeaua de recunoaștere facială și are rolul de a

transformă imaginea facială într-un vector de trăsături de dimensiune 512, care poate fi folosit ulterior în clasificare sau în evaluarea similarității.

Codul HRNet a fost preluat din repository-ul oficial InsightFace-tensorflow și a fost integrat în fișierul network.py, fără modificări. Modelul complet este construit de funcția hrnet_v2(), care compune rețeaua din trei componente principale: un bloc de inițializare (STEM), corpul rețelei (BODY) și un cap de ieșire (HEAD) care generează embedding-ul.

Funcția hrnet_v2() are următoarea formă:

```
def hrnet_v2(input_shape, output_size, width=18, trainable=True,
             kernel_regularizer=None, name="hrnetv2"):
    last_stage_width = sum([width * pow(2, n) for n in range(4)])

    inputs = keras.Input(input_shape, dtype=tf.float32)
    x = hrnet_stem(64, kernel_regularizer)(inputs)
    x = hrnet_body(width, kernel_regularizer)(x)
    outputs = hrnet_heads(input_channels=last_stage_width,
                          output_size=output_size,
                          kernel_regularizer=kernel_regularizer)(x)

    model = keras.Model(inputs=inputs, outputs=outputs,
                         name=name, trainable=trainable)
    return model
```

În primul rând, imaginea de intrare (cu dimensiunea $112 \times 112 \times 3$) trece prin blocul hrnet_stem(), care aplică o convolução 3×3 cu pas 2, urmată de normalizare batch și activare ReLU. Aceasta reduce rezoluția la 56×56 și pregătește datele pentru stadiile următoare. Codul blocului STEM este:

```
def hrnet_stem(filters=64, kernel_regularizer=None):
    stem_layers = [layers.Conv2D(filters, 3, 2, 'same',
                                 kernel_regularizer=kernel_regularizer),
                  layers.BatchNormalization(),
                  layers.Activation('relu')]

    def forward(x):
        for layer in stem_layers:
            x = layer(x)
        return x

    return forward
```

După acest pas, imaginea este procesată de funcția hrnet_body(), care implementează patru stadii principale: Stage 1 până la Stage 4. Fiecare stadiu adaugă o nouă ramură cu o rezoluție mai mică, dar păstrează în același timp toate ramurile existente active. În acest fel, modelul poate învăța simultan atât detalii locale (rezoluție înaltă), cât și trăsături globale (rezoluție joasă). Între ramuri se folosesc mecanisme de fuziune (FusionBlocks), care permit transferul de informație prin upsampling/downsampling.

Structura fiecărui stadiu este compusă din blocuri reziduale și bottleneck, preluate din fișierul resnet.py. Deși codul ResNet complet nu a fost folosit în acest proiect, anumite funcții (precum BottleneckBlock) au fost utilizate pentru a construi stadiile HRNet. Astfel, rețeaua păstrează stilul arhitectural descris teoretic în secțiunea 3.2.

După ce trece prin cele patru stadii, fiecare ramură este redimensionată la rezoluția maximă (56×56), apoi toate sunt concatenate. Această operație se face în funcția hrnet_heads(), care produce vectorul de embedding final:

```
def hrnet_heads(input_channels=56, output_size=256, kernel_regularizer=None):
    scales = [2, 4, 8]
    up_scale_layers = [layers.UpSampling2D((s, s)) for s in scales]
    concatenate_layer = layers.concatenate(axis=3)
    heads_layers = [layers.Conv2D(filters=input_channels, kernel_size=(1, 1),
                                  strides=(1, 1), padding='same',
                                  kernel_regularizer=kernel_regularizer),
                    layers.BatchNormalization(),
                    layers.Activation('relu'),
                    layers.GlobalAveragePooling2D(),
                    layers.Dense(output_size,
                                 kernel_regularizer=kernel_regularizer),
                    layers.BatchNormalization()]
    def forward(inputs):
        scaled = [f(x) for f, x in zip(up_scale_layers, inputs[1:])]
        x = concatenate_layer([inputs[0], scaled[0], scaled[1], scaled[2]])
        for layer in heads_layers:
            x = layer(x)
        return x
    return forward
```

Aici, ieșirile celor patru ramuri sunt scalate astfel încât să aibă aceeași rezoluție, apoi sunt concatenate pe axa canalelor. După o conoluție 1×1 , se aplică GlobalAveragePooling2D, iar apoi un strat dens care produce un vector de dimensiune 512 (embedding-ul final). Se aplică și BatchNormalization, pentru stabilizarea distribuției ieșirii.

Întreaga arhitectură HRNet este astfel concepută pentru a maximiza fidelitatea informației spațiale și pentru a produce embedding-uri robuste. Acestea vor fi folosite ulterior în antrenarea cu funcția de pierdere aleasă – fie Softmax, fie ArcFace – în funcție de modul de operare selectat.

6.3.2 Normalizarea L2 și stratul ArcFace – cod și corelare teoretică

Pentru a putea antrena modelul într-un mod specific recunoașterii faciale, am inclus în rețea un mecanism de proiecție pe sferă unitară, urmat de aplicarea unei margini unghiulare. Acest mecanism este parte esențială în abordarea ArcFace și are rolul de a produce embedding-uri bine separate între clase, dar compacte în interiorul fiecărei clase. În cod, această logică este implementată prin două componente principale: normalizarea L2 (L2Normalization) și stratul ArcLayer, ambele definite în fișierul network.py.

După ce HRNet produce un vector de embedding brut, acesta este proiectat pe sferă unitară prin normalizare L2. Această operație se face cu ajutorul clasei L2Normalization, care aplică funcția `tf.nn.l2_normalize()`:

```
class L2Normalization(layers.Layer):
    def __init__(self, axis=1, **kwargs):
        super().__init__(**kwargs)
        self.axis = axis

    def call(self, inputs, **kwargs):
        return tf.nn.l2_normalize(inputs, axis=self.axis)
```

Această normalizare asigură că fiecare embedding are normă unitară, adică toate punctele se află pe o sferă de rază 1 în spațiul latent. Astfel, distanțele dintre embedding-uri pot fi interpretate direct ca unghiuri între vectori, exact cum se explică și în capitolul teoretic al lucrării.

După normalizare, embedding-ul este transmis într-un strat special denumit ArcLayer. Acesta înlocuiește stratul de clasificare clasic cu unul care aplică un decalaj unghiular pentru clasa corectă, ceea ce forțează rețea să creeze separații mai clare între clase. Codul este următorul:

```
class ArcLayer(layers.Layer):
    def __init__(self, num_classes, regularizer=None, **kwargs):
        super().__init__(**kwargs)
        self.num_classes = num_classes
        self.kernel_regularizer = regularizer

    def build(self, input_shape):
        self.w = self.add_weight(
            name='W',
            shape=(input_shape[-1], self.num_classes),
            initializer='glorot_uniform',
            regularizer=self.kernel_regularizer,
            trainable=True
        )

    def call(self, inputs, **kwargs):
        inputs = tf.nn.l2_normalize(inputs, axis=1)
        w = tf.nn.l2_normalize(self.w, axis=0)
        cos_t = tf.matmul(inputs, w)
        return cos_t
```

Acest strat are două etape importante:

- normalizează atât vectorii de intrare (inputs), cât și ponderile stratului (self.w), pentru ca produsul scalar rezultat să fie un $\cos(\theta)$ – adică o măsură directă a unghiului dintre vector și clasa-țintă.
- returnează această valoare cosine pentru a fi folosită ulterior în calculul pierderii cu margine, adică în funcția ArcLoss (care va fi explicată detaliat în secțiunea 6.4).

Combinația dintre L2Normalization și ArcLayer are ca efect transformarea embedding-urilor într-un spațiu în care similaritatea este determinată strict unghiular. Această abordare aduce mai

multă stabilitate și generalizare în recunoaștere, deoarece nu se bazează pe valori absolute ale activărilor, ci doar pe direcțiile acestora în spațiul învățat.

În concluzie, acest mecanism completează logica teoretică ArcFace descrisă în capitolul anterior și este responsabil de obținerea embedding-urilor angulare, care sunt apoi folosite în evaluare sau inferență.

6.3.3 Configurarea modelului pentru antrenare: Softmax și ArcFace

Pentru a putea compara două moduri diferite de antrenare în recunoaștere facială, am implementat două configurații de model: una clasică, bazată pe clasificare cu Softmax, și una specializată, bazată pe funcția de pierdere ArcFace. Alegerea modului de antrenare se face direct din linia de comandă, prin argumentul --softmax, fără a fi nevoie să modific codul sursă.

Configurația modelului este realizată în fișierul train.py, în funcție de valoarea argumentului args.softmax. Cele două ramuri sunt prezentate în codul de mai jos:

```
if args.softmax:  
    print("Building training model with softmax loss...")  
    model = keras.Sequential([  
        keras.Input(input_shape),  
        base_model,  
        keras.layers.Dense(num_ids, kernel_regularizer=regularizer),  
        keras.layers.Softmax(),  
        name="training_model")  
    loss_fun = keras.losses.CategoricalCrossentropy()  
else:  
    print("Building training model with ARC loss...")  
    model = keras.Sequential([  
        keras.Input(input_shape),  
        base_model,  
        L2Normalization(),  
        ArcLayer(num_ids, regularizer)],  
        name="training_model")  
    loss_fun = ArcLoss()
```

În cazul în care se folosește varianta Softmax:

- Se adaugă un strat Dense cu un număr de ieșiri egal cu numărul de identități din dataset (num_ids).
- Se aplică o activare Softmax, care produce o distribuție de probabilitate.
- Funcția de pierdere este CategoricalCrossentropy, care compară distribuția ieșirii cu eticheta reală (one-hot encoded).

Această configurație este utilă pentru testare rapidă și învățare supravegheată clasică, unde se dorește ca modelul să învețe direct asocierile imagine–clasă.

În schimb, dacă se antrenează cu ArcFace:

- Se adaugă un strat L2Normalization, care proiectează embedding-ul pe sfera unitară.
- Apoi se aplică stratul ArcLayer, care produce similaritățile cosine între vectorii embedding și centrele claselor.
- Funcția de pierdere folosită este ArcLoss, care va aplica ulterior marja unghiulară asupra valorilor cosine (vezi secțiunea 6.4).

Această ramură este potrivită pentru scopul principal al proiectului, și anume antrenarea unui sistem de recunoaștere facială bazat pe embedding-uri discriminative. ArcFace a fost demonstrat teoretic ca fiind superior în scenarii de identificare și verificare facială, deoarece maximizează distanța unghiulară între clase.

Ambele variante folosesc același model de bază (base_model), adică HRNet, definit anterior prin funcția hrnet_v2(). Singura diferență constă în capul de rețea adăugat și în funcția de pierdere asociată. Acest design modular oferă flexibilitate și permite comutarea rapidă între cele două moduri de antrenare, fără a rescrie alte părți ale codului.

6.4 Funcția de pierdere ArcFace – explicație detaliată și cod

Funcția de pierdere joacă un rol esențial în învățarea embedding-urilor faciale. În cazul proiectului de față, am implementat o funcție de pierdere personalizată, denumită ArcLoss, care aplică principiile metodei ArcFace, discutate în capitolul teoretic. Aceasta introduce o margine unghiulară între clase, ceea ce forțează rețeaua să creeze spații latente în care vectorii embedding ai persoanelor diferite sunt mai bine separați.

Funcția este implementată în fișierul losses.py și extinde clasa keras.losses.Loss. Codul complet este următorul:

```
class ArcLoss(keras.losses.Loss):  
    def __init__(self, margin=0.5, scale=64.0, **kwargs):  
        super().__init__(**kwargs)  
        self.margin = margin  
        self.scale = scale  
    def call(self, y_true, y_pred):  
        # Convert labels from one-hot to indices  
        labels = tf.argmax(y_true, axis=1)  
        # Clip cosine similarities to avoid NaNs  
        cosine = tf.clip_by_value(y_pred, -1.0 + 1e-7, 1.0 - 1e-7)  
        theta = tf.acos(cosine)  
        # Adaugă marginea unghiulară pentru clasa corectă  
        target_logit = tf.cos(theta + self.margin)  
        # Înlocuiește logit-ul pentru clasa corectă cu cel modificat  
        target_one_hot = tf.one_hot(labels, depth=tf.shape(y_pred)[1])  
        logits = y_pred * (1 - target_one_hot) + target_logit * target_one_hot  
        # Aplică scaling și calculează pierderea softmax  
        logits *= self.scale  
        return keras.losses.categorical_crossentropy(y_true, logits, from_logits=True)
```

Comportamentul acestei funcții este următorul:

- Transformă etichetele de la format one-hot la format indexat (de exemplu, [0,0,1,0] devine 2).
- Prinde valorile cosine (ieșite din ArcLayer) în intervalul $-1,1$ pentru stabilitate numerică.
- Calculează unghiurile θ folosind funcția $\text{acos}()$.
- Adaugă o margine unghiulară m pentru clasa corectă și recalculează $\cos(\theta + m)$ – acest pas este cheia ideii ArcFace, și corespunde direct formulei prezentate în capitolul 3.
- Înlocuiește doar logit-ul pentru clasa corectă cu varianta modificată, păstrând celelalte valori neschimbate.
- Scalează toate logit-urile cu o constantă s (în mod uzual 64.0), pentru a amplifica diferențele și a face antrenarea mai eficientă.
- Aplică în final funcția de pierdere softmax cu logit-uri modificate.

Această funcție încurajează modelul să genereze embedding-uri care nu sunt doar corecte în clasificare, ci și mai bine distanțate angular între clase. Astfel, se obține un spațiu latent mai bine organizat, ceea ce este crucial pentru sarcini precum verificarea identității sau recunoaștere open-set.

În concluzie, ArcLoss este o extensie a pierderii clasice cross-entropy, care transformă problema intr-una unghiulară și impune un control geometric asupra felului în care sunt învățate embedding-urile faciale.

6.5 Procesul de antrenare – optimizare, validare, salvare

După configurarea modelului, urmează etapa esențială în care rețea neuronală învață să asocieze fețele din imagini cu identitățile corespunzătoare. În proiectul de față, întregul proces de antrenare este organizat în jurul clasei `TrainingSupervisor`, definită în fișierul `training_supervisor.py`. Aceasta centralizează toate operațiile legate de antrenare: propagare înainte, calculul pierderii, aplicarea gradientului, salvarea modelelor și logarea performanței.

Clasa este instanțiată în `train.py` după construirea modelului, astfel:

```
supervisor = TrainingSupervisor(model, optimizer, loss_fun,
                                  train_dataset=train_ds,
                                  val_dataset=val_ds,
                                  metrics=[train_acc_metric, val_acc_metric],
                                  log_dir=log_dir,
                                  ckpt_dir=checkpoint_dir,
                                  eval_freq=args.eval_freq,
                                  monitor=args.monitor,
                                  restore=args.restore_weights_only)
```

Modelul, optimizatorul și funcția de pierdere (`loss_fun`) sunt transmise ca argumente, împreună cu datele de antrenare și validare, directoarele pentru loguri și checkpoint-uri, precum și alți parametri controlați din linia de comandă (precum `eval_freq` sau `restore_weights_only`).

Optimizarea se face cu algoritmul Adam, configurat în train.py astfel:

```
optimizer = keras.optimizers.Adam(  
    learning_rate=args.learning_rate,  
    amsgrad=True,  
    epsilon=0.001,  
    clipvalue=1.0  
)
```

Aceste setări sunt alese pentru stabilitate numerică: amsgrad=True previne scăderi bruște ale ratei de învățare, epsilon mic ajută la evitarea divizărilor instabile, iar clipvalue limitează valoarea gradientului pentru a evita explodarea acestuia.

Odată inițializat, antrenarea efectivă se face apelând metoda train():

```
supervisor.train(epochs=args.epochs, initial_epoch=args.initial_epoch)
```

Această metodă gestionează bucla de învățare, care pentru fiecare epocă implică:

- iterarea prin batch-uri de imagini,
- propagarea înainte prin model,
- calculul pierderii,
- derivarea gradientului și actualizarea ponderilor rețelei.

Fragmențul din _train_step() responsabil cu această logică este:

```
with tf.GradientTape() as tape:  
    logits = self.model(x_batch, training=True)  
    loss = self.loss_fun(y_batch, logits) + sum(self.model.losses)  
    grads = tape.gradient(loss, self.model.trainable_weights)  
    self.optimizer.apply_gradients(zip(grads, self.model.trainable_weights))
```

Pe lângă actualizarea ponderilor, clasa actualizează și metricele curente, precum acuratețea, prin metodele `_update_metrics()` și `_reset_metrics()`.

Un alt aspect important este ajustarea dinamică a ratei de învățare. Aceasta este controlată de o funcție de tip „warm-up” urmată de o decădere liniară, care este apelată la fiecare pas:

```
new_lr = self.lr_schedule(step)  
self.optimizer.lr.assign(new_lr)
```

De asemenea, în timpul antrenării, modelul este validat periodic (nu doar la finalul epocii), folosind setul de validare. Frecvența validării este controlată cu argumentul `--eval_freq`. Dacă performanța pe validare depășește cea mai bună valoare înregistrată, modelul este salvat automat în fișierul de checkpoint:

```
if current > previous:  
    self.model.save_weights(checkpoint_path)
```

Toate valorile utile (loss, accuracy, rata de învățare) sunt logate în TensorBoard:

```
with self.writer.as_default():
    tf.summary.scalar("train_loss", train_loss, step=epoch)
```

La finalul antrenării, modelul este salvat într-o formă potrivită pentru inferență. Dacă s-a antrenat cu Softmax, se salvează modelul complet; în cazul ArcFace, se exportă doar baza de rețea (embedding extractor):

```
if args.export_only:
    if args.softmax:
        model.save(export_dir)
    else:
        supervisor.export(base_model, export_dir)
```

Această structură modulară și bine controlată face ca întreg procesul de antrenare să fie robust, scalabil și ușor de reluat sau extins. Prin folosirea clasei TrainingSupervisor, codul este organizat curat, iar componentele pot fi înlocuite fără a afecta întregul flux.

6.6 Evaluarea modelului – clasificare și analiză embedding

6.6.1 Evaluare clasică cu acuratețe și matrice de confuzie

Toate modelele antrenate în cadrul proiectului — inclusiv cel cu Softmax, cel cu ArcFace și cel antrenat în două faze (Softmax urmat de ArcFace) — au fost evaluate inițial în regim de clasificare multi-clasă. Scopul acestei evaluări este de a verifica dacă rețeaua poate atribui corect identitatea fiecărei imagini dintr-un set de testare complet separat, folosind doar predicția finală a modelului.

Procesul de evaluare este implementat în scriptul evaluate.py. Evaluarea se face prin rularea rețelei pe întregul set de testare, compararea etichetelor prezise cu cele reale și calculul scorurilor de clasificare.

Setul de test este încărcat din fișiere .tfrecord cu ajutorul funcției build_dataset():

```
test_ds = build_dataset(tfrecords=args.test_tfrecords,
                        batch_size=args.batch_size,
                        is_train=False)
```

Modelul exportat este încărcat cu funcția keras.models.load_model, iar în cazul în care a fost antrenat cu ArcFace, sunt înregistrate și straturile personalizate:

```
model = keras.models.load_model(args.model_path,
                                 custom_objects={'ArcLayer': ArcLayer,
                                                'L2Normalization': L2Normalization})
```

Apoi, pentru fiecare imagine din setul de test, modelul produce un vector de scoruri (logituri), iar clasa finală este determinată prin argmax:

```
y_pred, y_true = [], []  
  
for x_batch, y_batch in test_ds:  
    logits = model(x_batch, training=False)  
    y_pred.extend(np.argmax(logits.numpy(), axis=1))  
    y_true.extend(np.argmax(y_batch.numpy(), axis=1))  
  
accuracy = accuracy_score(y_true, y_pred)  
print("Accuracy:", accuracy)
```

Această metodă de evaluare este valabilă și aplicabilă chiar dacă modelul a fost antrenat cu funcția de pierdere ArcFace. În acest caz, deși scopul principal al modelului este de a genera embedding-uri robuste, el este capabil și să producă direct o predicție de clasă prin similaritatea cosine în stratul final. Astfel, și aceste modele pot fi comparate într-un mod unitar cu cele antrenate cu Softmax.

Pentru o analiză mai detaliată a erorilor de clasificare, a fost generată și o matrice de confuzie completă, de dimensiune 100×100 (corespunzătoare celor 100 de clase din setul de testare), folosind funcția confusion_matrix din sklearn:

```
cm = confusion_matrix(y_true, y_pred)
```

Aceasta este afișată ca heatmap cu ajutorul bibliotecii Seaborn:

```
plt.figure(figsize=(12, 10))  
sns.heatmap(cm, cmap='Blues')  
plt.title("Confusion Matrix")  
plt.xlabel("Predicted")  
plt.ylabel("Actual")  
plt.tight_layout()  
plt.show()
```

Totuși, o astfel de matrice completă este greu de interpretat din cauza numărului mare de clase. Pentru a evidenția zonele cele mai problematice, scriptul selectează automat cele mai frecvente 20 de persoane confundate și generează un heatmap normalizat, cu valori exprimate procentual:

```
_plot_top_confusions_heatmap(cm, top_n=20, fname="heatmap_top_confusions.png")
```

Această abordare permite o vizualizare clară a celor mai dificile cazuri de clasificare, fără a fi nevoie să se analizeze întreaga matrice de 100×100 .

Pentru o analiză vizuală completă, scriptul salvează automat exemple ilustrative pentru două situații distincte:

- clasificări greșite: pentru fiecare confuzie selectată, sunt afișate imaginea reală și o imagine reprezentativă din clasa prezisă greșit, folosind funcția `plot_confusion_examples()`,
- clasificări corecte: sunt selectate aleator două imagini diferite din aceeași clasă care au fost etichetate corect, pentru a evidenția coerența vizuală a embedding-urilor. Acestea sunt afișate cu funcția `plot_correct_examples()`.

Această dublă vizualizare contribuie la o evaluare mai completă a comportamentului rețelei, evidențiind atât limitele în cazurile de confuzie, cât și consistența predicțiilor reușite. Astfel, evaluarea clasică oferă o imagine de ansamblu valoroasă asupra performanței în regim de clasificare, iar în combinație cu analiza embedding-urilor (prezentată în secțiunea următoare), permite o validare riguroasă și echilibrată a întregului sistem.

6.6.2 Evaluare embedding cu similaritate cosine și ROC

Pentru evaluarea embedding-urilor generate de rețea, am aplicat o metodă specifică recunoașterii faciale, care nu se bazează pe clasificare directă, ci pe măsurarea similarității dintre vectorii obținuți. Acest tip de evaluare a fost aplicat pentru toate cele trei modele antrenate: atât cel cu Softmax, cât și cele cu ArcFace (antrenare completă sau în două faze). Analiza s-a realizat cu ajutorul scriptului `evaluate.py`, care conține un modul dedicat pentru compararea embedding-urilor în perechi pozitive (aceeași persoană) și negative (persoane diferite).

Pentru fiecare model exportat, imaginile de test au fost parcuse în perechi. Pentru fiecare pereche, embedding-urile generate au fost comparate folosind similaritatea cosine, care oferă un scor între -1 și 1. Scorurile ridicate indică asemănare mare între fețe, iar cele scăzute indică diferențe semnificative. Această metodă reflectă direct geometria învățată în spațiul latent, în special în cazul antrenării cu ArcFace, unde vectorii sunt normalizați L2 și proiecțiile sunt angulare.

Pe baza acestor scoruri, a fost generată curba ROC (Receiver Operating Characteristic), care evidențiază performanța modelului în a distinge perechi pozitive de cele negative, la diferite praguri de decizie. Curba ROC a fost construită folosind funcțiile `roc_curve` și `auc` din `sklearn`:

```
fpr, tpr, _ = roc_curve(true_labels, sims)
roc_auc = auc(fpr, tpr)
```

Aria de sub această curbă (AUC) a fost folosită ca metrică principală de performanță. Cu cât AUC este mai apropiată de 1.0, cu atât modelul reușește mai bine să diferențieze persoanele între ele. Rezultatele obținute confirmă că modelele antrenate cu ArcFace (inclusiv varianta cu pre-antrenare Softmax) generează embedding-uri mai robuste și mai bine structurate.

Pentru a analiza distribuția scorurilor de similaritate, a fost generată și o histogramă cosine care separă perechile pozitive de cele negative. Dacă cele două distribuții sunt bine separate, înseamnă că embedding-urile sunt discriminative. În caz contrar, un prag de decizie devine mai greu de ales.

```
_plot_histogram(pos_sims, neg_sims, fname="cosine_histogram.png")
```

În plus, a fost calculat și un prag optim de decizie, corespunzător punctului în care se maximizează acuratețea pe setul de perechi. Deși această evaluare este similară unei sarcini de clasificare binară, scopul ei principal este de a arăta cât de coherent este spațiul embedding.

Pentru interpretarea vizuală a greșelilor, a fost generată o matrice de confuzie pe baza claselor din setul de test. Totuși, întrucât setul conține 100 de persoane, o matrice completă 100×100 este dificil de interpretat vizual. Din acest motiv, scriptul a extras automat cele mai frecvente 20 de clase confundate, iar pentru acestea s-a generat un heatmap normalizat:

```
_plot_top_confusions_heatmap(cm, top_n=20, fname="heatmap_top_confusions.png")
```

Această metodă permite o analiză concentrată asupra zonelor în care apar cele mai multe erori și evidențiază tiparele de confuzie între persoane cu trăsături similare.

În concluzie, evaluarea bazată pe embedding-uri a oferit o perspectivă mult mai detaliată asupra capacitatei modelului de a învăța un spațiu latent bine organizat. Prin folosirea curbei ROC, a scorurilor cosine și a vizualizărilor asociate, am putut evalua nu doar performanța brută, ci și calitatea reprezentării interne învățate de rețea.

Aceste concluzii vor fi completeate și susținute în capitolul următor, unde vor fi prezentate în detaliu rezultatele experimentale și comparația între cele trei configurații testate.

7 Rezultate experimentale

7.1 Setul de date

Pentru antrenarea și evaluarea rețelei de recunoaștere facială dezvoltate în cadrul acestei lucrări, s-a utilizat o variantă adaptată a setului de date MS1M-ArcFace, un subset curățat și preprocesat al bazei MS-Celeb-1M, oferit de echipa InsightFace [Deng2019].

MS1M-ArcFace este disponibil public în format structurat pe clase, conținând aproximativ 85.000 de identități și peste 5.8 milioane de imagini. Datorită dimensiunii ridicate a datasetului complet și a limitărilor hardware ale sistemului utilizat (antrenarea fiind efectuată pe un laptop personal, fără acces la GPU de mare capacitate), s-a optat pentru extragerea unui subset reprezentativ format din 100 de persoane.

7.1.1 Structurarea subsetului

Subsetul extras conține un total de 7.208 imagini corespunzătoare celor 100 de identități selectate aleatoriu, respectând o distribuție echilibrată a imaginilor pe clasă. Aceste imagini au fost împărțite în trei subseturi funcționale:

Tabelul 1. Structura subsetului de imagini utilizat pentru antrenare, validare și testare.

Subset	Număr imagini	Procent din total	Scop principal
Train	4.999	~69.35%	Antrenarea rețelei neuronale
Validation	1.030	~14.29%	Evaluare periodică în timpul antrenării
Test	1.179	~16.36%	Evaluare finală (open-set)

Imaginiile au fost repartizate folosind o metodă stratificată, astfel încât fiecare identitate să fie proporțional reprezentată în toate cele trei subseturi. Această împărțire asigură coerenta evaluării și previne dezechilibrul dintre clase.

7.1.2 Format și organizare

Imaginiile sunt stocate în directoare corespunzătoare numelui clasei (numele persoanei), iar această organizare a fost convertită ulterior în format TFRecord compatibil cu TensorFlow. Astfel, fiecare eșantion include atât imaginea, cât și eticheta asociată, derivată direct din numele folderului.

Exemplu de structură:

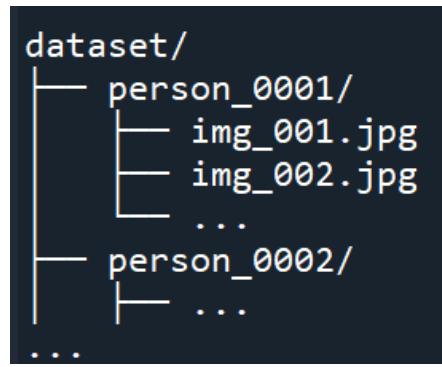


Figura 5. Exemplu de structură dataset

În figura de mai jos sunt prezentate câteva exemple de imagini din subsetul utilizat, corespunzătoare primelor cinci clase (persoanele 0–4). Se observă variabilitatea naturală în iluminare, unghi și calitate, care influențează dificultatea sarcinii de recunoaștere.



Figura 6. Exemple de imagini din subsetul utilizat (persoanele 0–4)

7.1.3 Convertirea în format TFRecord

Pentru o eficiență sporită în timpul antrenării, imaginile și etichetele au fost convertite din structură de directoare în format binar de tip TFRecord. Acest format reduce timpul de acces la date și permite o încărcare secvențială optimizată în GPU, fiind utilizat frecvent în proiecte TensorFlow.

Fișierele .record generate sunt:

- train.record – conține toate imaginile de antrenare și etichetele corespunzătoare,
- val.record – folosit exclusiv pentru evaluarea periodică în timpul antrenării,
- test.record – destinat evaluării finale după antrenare.

În plus, fișierul test_pairs_lfw.bin conține perechi de imagini (pozitive și negative) utilizate în etapa de testare pentru analiza distanțelor și a scorurilor de similaritate cosine, conform formatului setului de validare LFW [Huang2008].

7.1.4 Considerații practice

Decizia de a utiliza doar 100 de persoane din datasetul complet MS1M-ArcFace a fost determinată de constrângerile legate de timp și resurse de calcul. Antrenarea întregului set ar fi necesitat perioade lungi de procesare și echipamente GPU avansate, ceea ce nu a fost posibil în condițiile disponibile. Chiar și cu subsetul ales, s-au obținut rezultate relevante pentru analiza

comparativă a metodelor Softmax și ArcFace, aşa cum se va prezenta în secțiunile următoare.

7.2 Metodologia experimentelor

Pentru a evalua performanța arhitecturii implementate, au fost definite trei scenarii de antrenare și testare, fiecare având la bază o configurație diferită a funcției de pierdere și a capului de rețea. Obiectivul principal al experimentelor a fost analiza comparativă între abordarea tradițională de clasificare (Softmax), metoda bazată pe separabilitate angulară (ArcFace) și o strategie combinată care îmbină stabilitatea primeia cu discriminabilitatea celei de-a doua.

7.2.1 Scenariile experimentale

Cele trei scenarii de antrenare au fost următoarele:

- Model 1 – Softmax: rețeaua a fost antrenată timp de 30 de epoci folosind pierderea CrossEntropyLoss și capul de rețea Softmax.
- Model 2 – ArcFace: același model de bază a fost antrenat timp de 30 de epoci exclusiv cu pierderea ArcFace și capul de tip ArcMarginProduct.
- Model 3 – Softmax + ArcFace: antrenare în două faze: inițial 7 epoci cu Softmax pentru stabilizare, urmate de 23 epoci cu ArcFace pentru rafinarea embedding-urilor.

Antrenările au fost realizate pe subsetul de 100 de identități, conform structurii descrise în secțiunea 5.1. Hyperparametrii și comenzile folosite sunt documentate în capitolul anterior (Implementarea soluției).

7.2.2 Evaluare și metriki

Pentru testare, s-a utilizat un set de 1.179 imagini complet separat de cel de antrenare și validare. Evaluarea a fost realizată în două moduri complementare:

Clasificare directă: pentru fiecare imagine, modelul a prezis clasa corespunzătoare (identitatea). S-a calculat acuratețea (procentul de predicții corecte) și pierderea totală pe setul de testare.

Evaluare avansată a embedding-urilor: s-au extras embedding-urile generate de fiecare model și s-au analizat 1.000 de perechi de imagini (500 pozitive, 500 negative). Fiecare pereche a fost evaluată folosind similaritatea cosine, iar distribuția scorurilor a fost folosită pentru:

- construirea histogramelor (separare pozitive vs. negative),
- generarea curbei ROC (Receiver Operating Characteristic) și calculul ariei sub curbă (AUC).

Evaluarea a fost realizată cu ajutorul scriptului evaluate.py, care include atât partea de clasificare, cât și compararea embedding-urilor pe bază de cosine distance. Acest cod permite testarea uniformă a tuturor celor trei modele, fără modificări semnificative.

Toate testele au fost efectuate în aceleași condiții hardware și folosind aceleași seturi de date. Pentru fiecare model, s-a folosit versiunea salvată cu cele mai bune performanțe pe setul de validare (checkpoint optim). Acest lucru a permis o comparație echitabilă între scenarii, eliminând variațiile externe.

7.3 Rezultate obținute

Evaluarea celor trei modele a furnizat rezultate relevante atât în termeni de acuratețe de clasificare, cât și în ceea ce privește calitatea embedding-urilor în sarcini de verificare facială.

Tabelul următor sintetizează performanțele obținute pe setul de testare:

Tabelul 2 . Rezultate obținute pentru fiecare model pe setul de testare.

Model	Acuratețe (%)	Pierdere (loss)	AUC ROC
Softmax	89.20	8.5566	0.9920
ArcFace	89.46	8.6397	0.9930
Softmax + ArcFace	88.44	9.7116	0.9925

Astfel, în toate cele trei cazuri, modelele reușesc să diferențieze eficient între perechile de imagini, iar valorile AUC confirmă o capacitate de discriminare robustă.

Curbele ROC aferente fiecărui model sunt ilustrate în figura următoare:

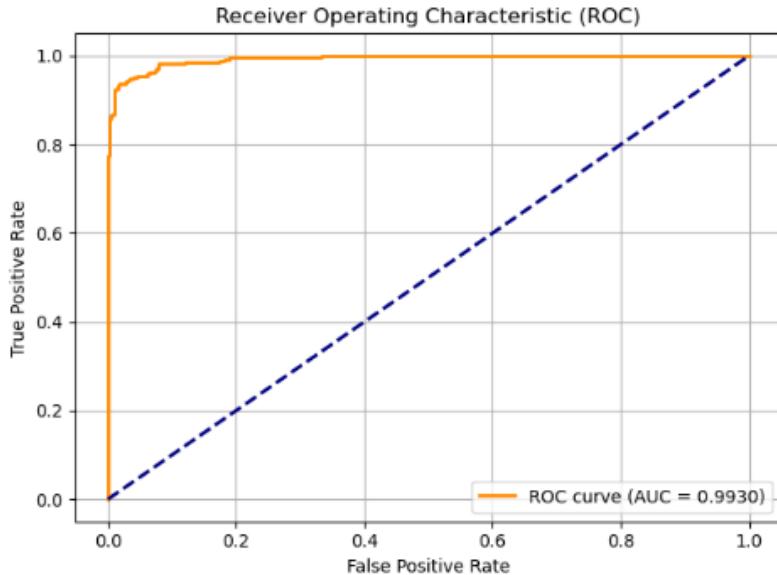


Figura 7. Curba ROC pentru modelul ArcFace

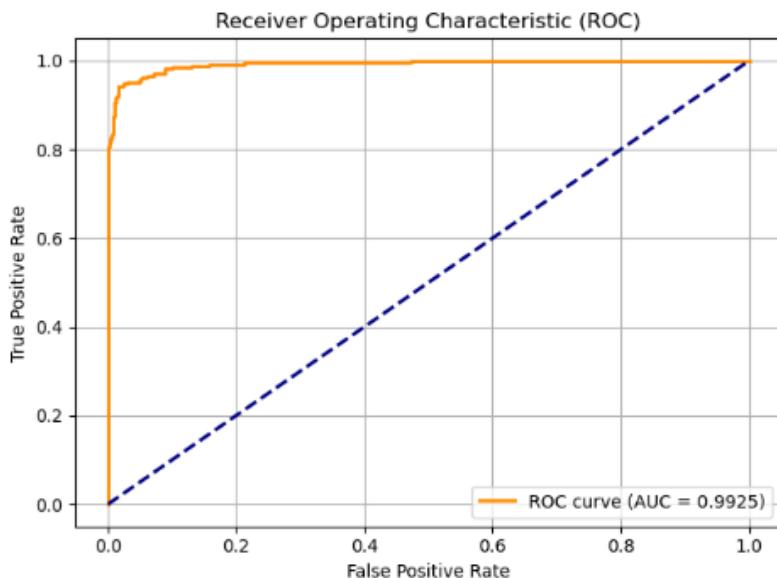


Figura 8. Curba ROC pentru modelul Softmax

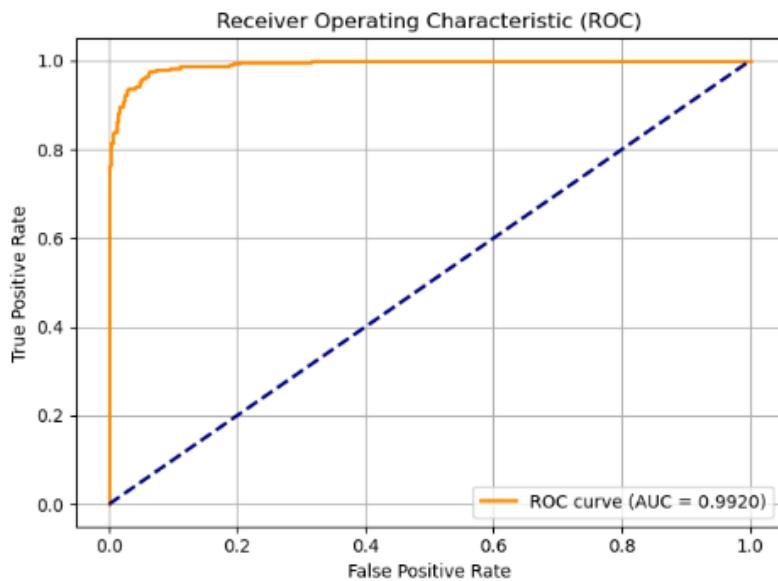


Figura 9. Curba ROC pentru modelul Softmax+ArcFace

Distribuțiile scorurilor cosine sunt prezentate în următoarele histograme:

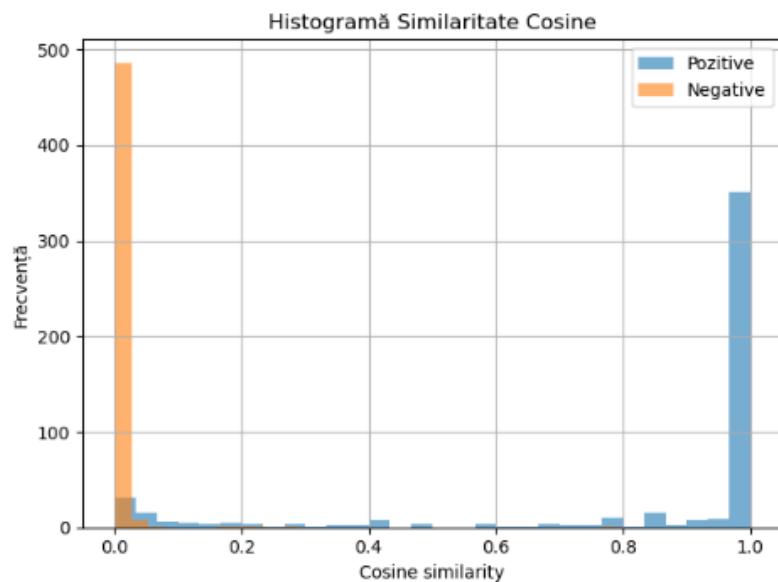


Figura 10. Histogramă similaritate cosine – model ArcFace

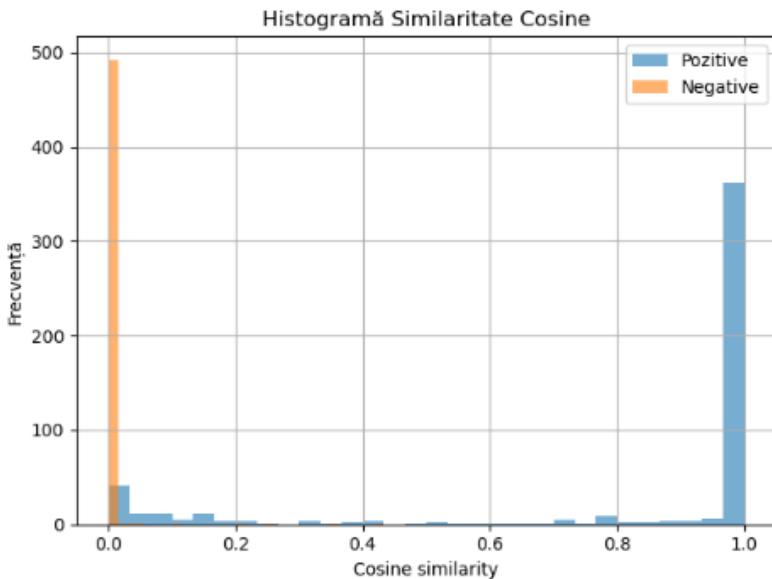


Figura 11. Histogramă similaritate cosine – model Softmax

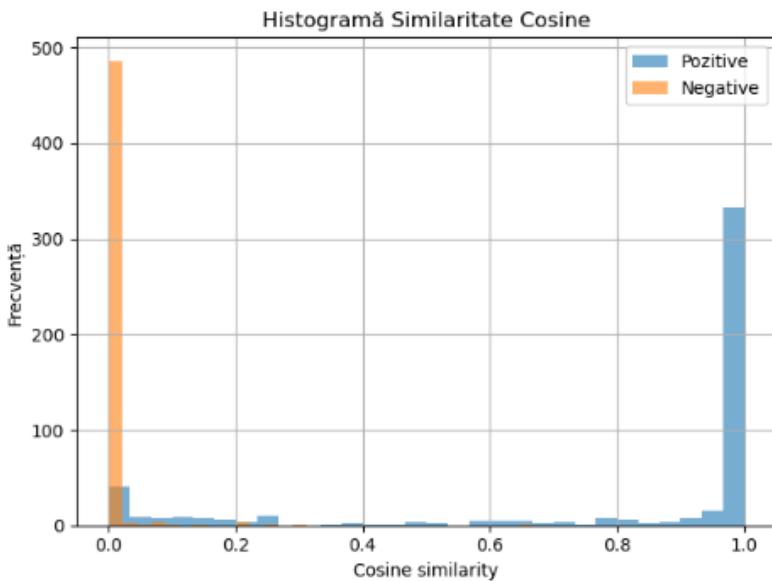


Figura 12. Histogramă similaritate cosine – model Softmax+ArcFace

În plus față de aceste metriki, în cadrul evaluării au fost introduse trei funcționalități suplimentare pentru o analiză vizuală detaliată a performanței modelului:

- Heatmapul confuziilor: pe baza matricei de confuzie, a fost generat automat un heatmap care evidențiază cele mai frecvente 20 de perechi de clase confundate de model. Acest instrument vizual este util pentru a identifica identități asemănătoare sau imagini problematice (ex. unghiuri, iluminare slabă).
- Exemple vizuale ale erorilor: pentru cazurile în care modelul a făcut confuzii semnificative, au fost salvate perechi de imagini formate din imaginea de test și imaginea din clasa greșit prezisă cu cea mai mare similaritate cosine. Acestea oferă o perspectivă practică asupra limitărilor sistemului și ajută la înțelegerea cauzelor confuziilor.

- Exemple vizuale ale clasificărilor corecte: pentru a completa analiza, au fost selectate și perechi de imagini care au fost clasificate corect, cu un scor de similaritate cosine ridicat. Aceste exemple evidențiază robustețea embedding-urilor și capacitatea rețelei de a generaliza în prezența variațiilor de unghi, iluminare sau expresie facială.

Cele trei tipuri de vizualizări sunt ilustrate în imaginile următoare:

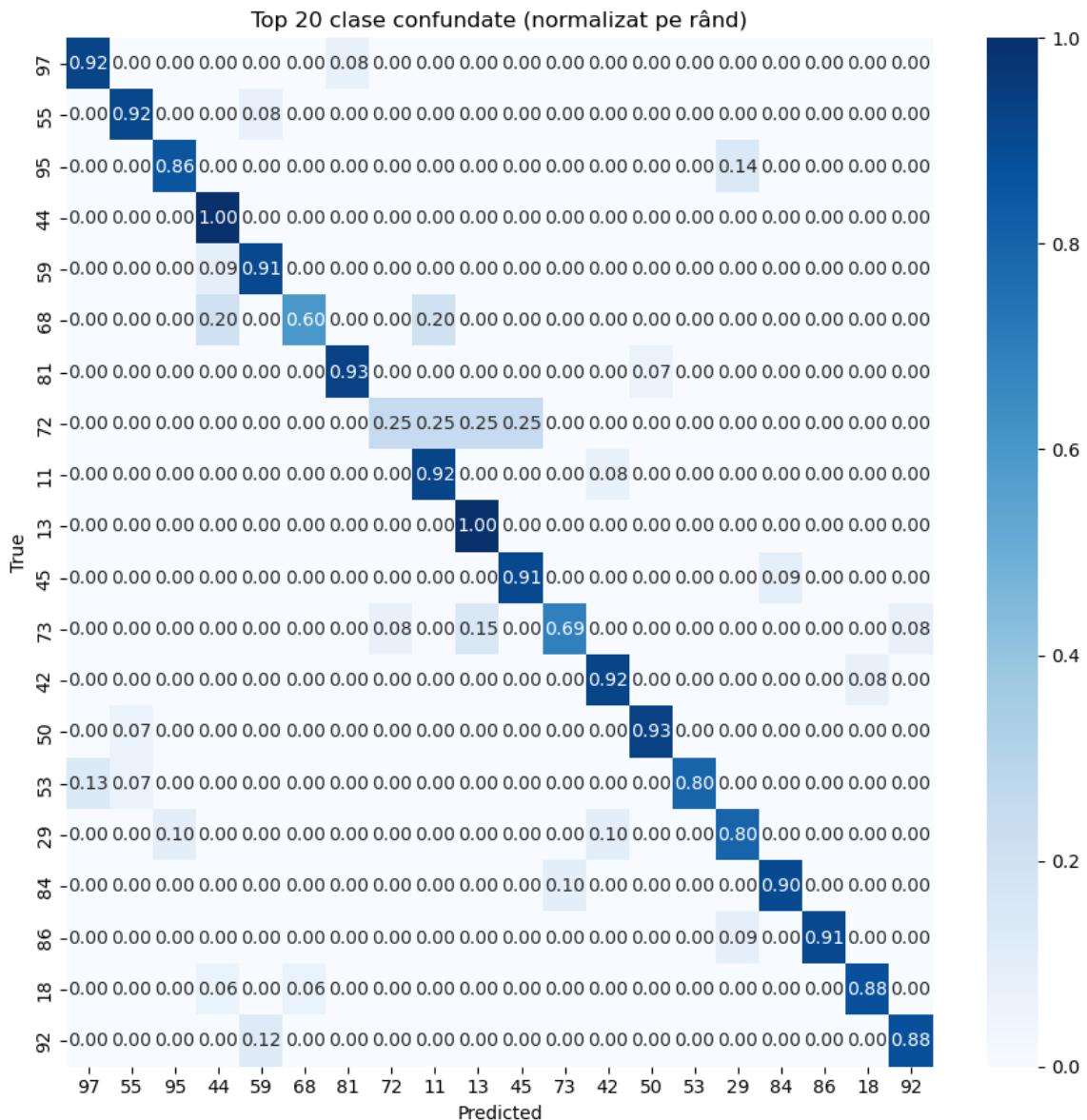


Figura 13. Heatmap al celor mai frecvent confundate perechi de clase

Confuzie #1

Corect: 16

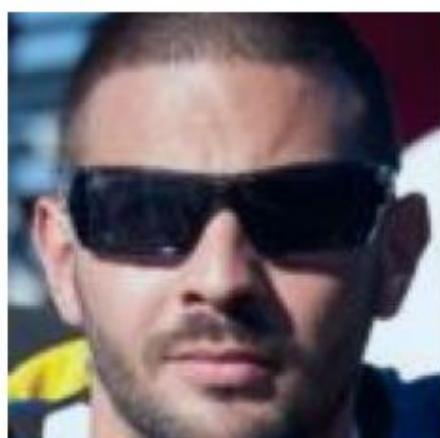


Prezicere: 34



Confuzie #2

Corect: 45

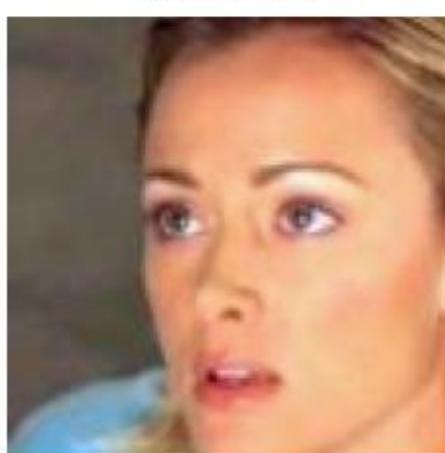


Prezicere: 33



Confuzie #3

Corect: 40



Prezicere: 4



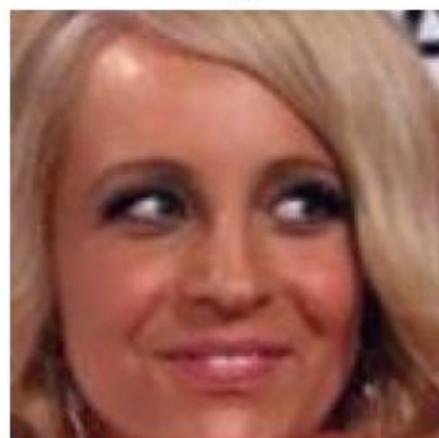
Figura 14. Exemple vizuale de clasificări greșite (stânga: imagine test; dreapta: clasă prezisă greșit cu scor ridicat)

Predicție corectă #1

Corect: 38



Alt exemplu: 38

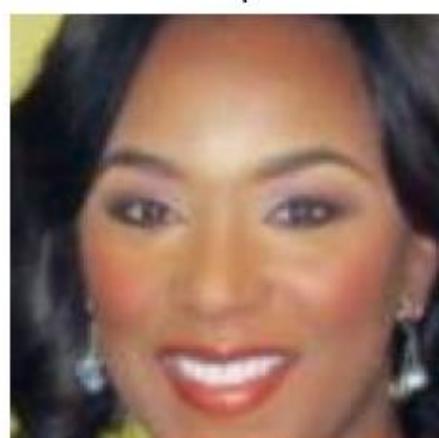


Predicție corectă #2

Corect: 25



Alt exemplu: 25



Predicție corectă #3

Corect: 82



Alt exemplu: 82



Figura 15. Exemple vizuale de clasificări corecte (stânga: imagine test; dreapta: imagine din aceeași clasă, corect prezisă)

7.4 Interpretarea comparativă a rezultatelor

Rezultatele obținute în urma evaluării experimentale confirmă faptul că toate cele trei modele analizate (Softmax, ArcFace și combinația Softmax + ArcFace) oferă performanțe ridicate și relativ apropiate în contextul unui set de date redus, curat și echilibrat. Acuratețile sunt comparabile, iar valorile AUC ale curbelor ROC depășesc 0.99 în toate cazurile, indicând o capacitate excelentă de discriminare între perechi pozitive și negative.

Histogramele cosine obținute pentru cele trei modele confirmă, de asemenea, acest comportament similar. Perechile pozitive tind să aibă o similaritate apropiată de +1, iar cele negative sunt concentrate în jurul valorii 0. Astfel, diferențele dintre metodele de antrenare sunt mai puțin vizibile în contextul subsetului utilizat.

Pentru o înțelegere mai profundă a comportamentului rețelei, au fost adăugate două funcționalități suplimentare în codul de evaluare:

- Heatmap-ul celor mai confundate 20 de clase, ilustrat în Figura 13, evidențiază cu precizie perechile de identități între care apar cele mai frecvente erori. De exemplu, clasele 45 și 33, sau 40 și 4, au fost confundate de mai multe ori, indicând o suprapunere vizuală semnificativă în spațiul de embedding.
- Exemple vizuale ale confuziilor, prezentate în Figura 14, oferă o perspectivă concretă asupra erorilor. Aceste imagini arată, pentru fiecare caz selectat, fotografia de test (eticheta corectă) și o imagine reprezentativă din clasa prezisă greșit cu cel mai mare scor de similaritate cosine. Se observă că majoritatea confuziilor apar între fețe cu expresii similare, accesoriu (ochelari), unghiuri neobișnuite sau iluminare nefavorabilă.

Pentru completarea acestei analize și pentru a evidenția exemplele de clasificări reușite, au fost adăugate și exemple vizuale de predicții corecte, prezentate în Figura 15. În aceste cazuri, imaginea de test a fost încadrată corect într-o clasă, iar alături este afișat un alt exemplu din aceeași clasă, pentru a evidenția coerența vizuală în spațiul de embedding. Aceste imagini ilustrează scenarii în care modelul a identificat cu succes asemănările relevante, în ciuda variațiilor de iluminare sau expresie.

Aceste reprezentări vizuale au un dublu rol: pe de o parte confirmă calitatea generală a modelului (deoarece majoritatea confuziilor sunt explicabile, iar clasificările corecte sunt consecvente), iar pe de altă parte oferă puncte de plecare pentru îmbunătățirea datasetului sau a procesului de aliniere a fețelor.

Totuși, trebuie menționat că aceste rezultate nu anulează avantajele teoretice ale funcției de pierdere ArcFace. Dimpotrivă, ArcFace este concepută pentru a oferi separabilitate superioară în spațiul de embedding-uri în contexte mult mai dificile, cu mii de identități, imagini nealiniiate sau condiții nefavorabile. În astfel de scenarii, marja unghiulară devine esențială pentru învățarea unor reprezentări robuste și bine separate.

Prin urmare, chiar dacă în cadrul acestui experiment cele trei modele oferă performanțe apropiate, în aplicații reale sau la scară mare, ArcFace rămâne abordarea preferată datorită generalizării mai bune și a controlului superior asupra distribuției embedding-urilor.

8 Concluzii

Lucrarea de față a avut ca obiectiv proiectarea, implementarea și evaluarea unui sistem modern de recunoaștere facială, construit pe baza rețelelor neuronale convoluționale și a unor funcții avansate de pierdere. Arhitectura principală utilizată a fost HRNet, recunoscută pentru capacitatea de a păstra rezoluții înalte pe tot parcursul rețelei, aceasta fiind combinată cu două funcții de pierdere Softmax și ArcFace pentru a asigura atât stabilitatea antrenării, cât și o separare optimă a claselor în spațiul embedding. Înțregul sistem a fost gândit modular, cu accent pe flexibilitate și pe posibilitatea de a testa riguros mai multe scenarii experimentale.

Evaluările au fost efectuate pe un subset extras din MS1M-ArcFace, compus din 100 de identități și 7.208 imagini, organizat pe trei subseturi (antrenare, validare și testare). Au fost analizate trei scenarii de antrenare: model clasic Softmax, model ArcFace pur și un regim hibrid Softmax + ArcFace în două faze. Rezultatele obținute au evidențiat performanțe consistente, cu valori de acuratețe în jur de 89% și arii sub curba ROC (AUC) de peste 0,99, confirmând o capacitate ridicată de discriminare a rețelei.

Pe lângă aceste evaluări clasice, lucrarea a propus două instrumente suplimentare de analiză: (1) un heatmap al celor mai frecvent confundate clase, util pentru identificarea perechilor de identități similare, respectiv (2) vizualizări explicite ale erorilor, prin perechi de imagini (test și clasă prezisă greșit) care au obținut scoruri de similaritate ridicate. Aceste contribuții au permis o înțelegere aprofundată a modului în care embedding-urile învață structura datelor și a limitărilor inerente în prezența unor imagini cu dificultăți (unghiuri nefavorabile, ochelari, iluminare slabă).

Din punct de vedere tehnic, printre contribuțiiile originale se remarcă structurarea modulară a codului, integrarea unui mecanism dinamic de alegere a funcției de pierdere prin argumente de linie de comandă, implementarea unei strategii de antrenare în două faze (Softmax + ArcFace), extinderea procedurilor de checkpointing pentru o reluare robustă a experimentelor, precum și realizarea unui script de evaluare avansată care generează curbe ROC, histograme cosine, heatmapuri de confuzii și exemple vizuale de clasificări greșite sau corecte. Înțregul sistem a fost testat în condiții reale, pe un echipament cu resurse limitate, reușind totuși să atingă performanțe competitive.

Este important de semnalat totuși o limitare: dimensiunea relativ redusă a subsetului utilizat nu a permis evidențierea completă a avantajelor ArcFace în contexte mult mai complexe, cu mii de clase și variații severe. De asemenea, sistemul nu a fost extins în această etapă pentru funcționare în timp real sau pentru validare pe baze de date externe.

Perspectivele de continuare a acestei lucrări sunt promițătoare și vizează: integrarea unei camere video pentru recunoaștere facială live, adaptarea modelului pentru dispozitive embedded prin cuantizare și compresie, testarea pe baze de date publice externe (de exemplu LFW), antrenarea pe imagini necurate sau augmentate pentru creșterea robustei și integrarea unor metode avansate de aliniere facială.

În concluzie, lucrarea demonstrează eficiența utilizării arhitecturii HRNet în combinație cu funcția de pierdere ArcFace, în cadrul unui sistem flexibil și modular de recunoaștere facială. Rezultatele experimentale confirmă robustețea metodei propuse, iar caracterul extensibil al implementării oferă un fundament solid pentru dezvoltări ulterioare orientate către aplicații practice, complexe și scalabile.

9 Bibliografie

1. [Sun2019] Sun, K., Xiao, B., Liu, D., & Wang, J., Deep High-Resolution Representation Learning for Human Pose Estimation, Proceedings of CVPR, IEEE, Long Beach (USA), June 2019.
2. [Deng2019] Deng, J., Guo, J., Xue, N., & Zafeiriou, S., ArcFace: Additive Angular Margin Loss for Deep Face Recognition, Proceedings of CVPR, IEEE, Long Beach (USA), June 2019.
3. [Jain2011] Jain, A. K., Ross, A., & Nandakumar, K., Introduction to Biometrics, Springer, New York (USA), 2011.
4. [Turk1991] Turk, M. A., & Pentland, A. P., Eigenfaces for recognition, Journal of Cognitive Neuroscience, Vol. 3(1), 1991, pp. 71–86.
5. [Belhumeur1997] Belhumeur, P. N., Hespanha, J. P., & Kriegman, D. J., Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 19(7), July 1997, pp. 711–720.
6. [Ahonen2006] Ahonen, T., Hadid, A., & Pietikäinen, M., Face Description with Local Binary Patterns: Application to Face Recognition, IEEE TPAMI, Vol. 28(12), Dec. 2006, pp. 2037–2041.
7. [Krizhevsky2012] Krizhevsky, A., Sutskever, I., & Hinton, G. E., ImageNet Classification with Deep Convolutional Neural Networks, NeurIPS, Vol. 25, 2012, pp. 1097–1105.
8. [Wang2018] Wang, H., Wang, Y., Zhou, Z., Ji, X., Gong, D., Zhou, J., Li, Z., & Liu, W., CosFace: Large Margin Cosine Loss for Deep Face Recognition, Proceedings of CVPR, IEEE, Salt Lake City (USA), June 2018.
9. [Huang2008] Huang, G. B., Ramesh, M., Berg, T., & Learned-Miller, E., Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments, Technical Report 07-49, University of Massachusetts, 2008.
10. [Guo2021] Guo, Y., Zhang, L., Hu, Y., He, X., & Gao, J., InsightFace: An open-source 2D&3D deep face analysis toolbox, GitHub, [Online], 2021.
<https://github.com/deepinsight/insightface>
11. [Schroff2015] Schroff, F., Kalenichenko, D., & Philbin, J., FaceNet: A Unified Embedding for Face Recognition and Clustering, Proceedings of CVPR, IEEE, Boston (USA), June 2015.
12. [Liu2016] Liu, W., Wen, Y., Yu, Z., & Yang, M., Large-Margin Softmax Loss for Convolutional Neural Networks, Proceedings of ICML, PMLR, New York (USA), June 2016.
13. [Ranjan2017] Ranjan, R., Castillo, C. D., & Chellappa, R., L2-Constrained Softmax Loss for Discriminative Face Verification, arXiv preprint, 2017.
<https://arxiv.org/abs/1703.09507>
14. [Nair2010] Nair, V., & Hinton, G. E., Rectified Linear Units Improve Restricted Boltzmann Machines, Proceedings of ICML, Haifa (Israel), June 2010.
15. [Kingma2015] Kingma, D. P., & Ba, J. L., Adam: A Method for Stochastic Optimization, ICLR, San Diego (USA), 2015. <https://arxiv.org/abs/1412.6980>
16. [Smith2017] Smith, L. N., Cyclical Learning Rates for Training Neural Networks, IEEE Winter Conference on Applications of Computer Vision (WACV), Santa Rosa (USA), 2017.
17. [Goyal2017] Goyal, P., et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, Facebook AI Research, arXiv preprint, 2017. <https://arxiv.org/abs/1706.02677>

18. [Bengio2012] Bengio, Y., Practical Recommendations for Gradient-Based Training of Deep Architectures, Lecture Notes in Computer Science, Springer, Berlin (Germany), 2012.
19. [Abadi2016] Abadi, M., et al., TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, Google Research, arXiv preprint, 2016. <https://arxiv.org/abs/1603.04467>
20. [Hanley1982] Hanley, J. A., & McNeil, B. J., The Meaning and Use of the Area under a ROC Curve, Radiology, Vol. 143(1), 1982, pp. 29–36.
21. [Grother2019] Grother, P., Ngan, M., & Hanaoka, K., Face Recognition Vendor Test (FRVT) Part 3: Demographic Effects, NIST Interagency Report 8280, 2019.
22. [Zhou2021] Zhou, Z., et al., Evaluating the Interpretability of Deep Learning Models: A Face Recognition Case Study, Proceedings of ECCV, Springer, Munich (Germany), 2021.
23. [DoshiVelez2017] Doshi-Velez, F., & Kim, B., Towards a Rigorous Science of Interpretable Machine Learning, arXiv preprint, 2017. <https://arxiv.org/abs/1702.08608>
24. [Barcic2023] Barčić, E., Grd, P., & Tomičić, I., Convolutional Neural Networks for Face Recognition: A Systematic Literature Review, arXiv preprint, 2023. <https://arxiv.org/abs/2303.02187>
25. [Guo2016] Guo, Y., Zhang, L., Hu, Y., He, X., & Gao, J., MS-Celeb-1M: A Dataset and Benchmark for Large-Scale Face Recognition, Proceedings of ECCV, Springer, Amsterdam (Netherlands), October 2016.
26. [InsightFace2020] InsightFace, MS1M-ArcFace Dataset (cleaned), GitHub repository, [Online], 2020. <https://github.com/deepinsight/insightface/tree/master/recognition/datasets>
27. [Ronghuaiyang2020] Ronghuaiyang, arcface-pytorch (TensorFlow reimplementation), GitHub repository, [Online], 2020. <https://github.com/ronghuaiyang/arcface-pytorch>
28. [Goodfellow2016] Goodfellow, I., Bengio, Y., & Courville, A., Deep Learning, MIT Press, Cambridge (USA), 2016.
29. [OShea2015] O'Shea, K., & Nash, R., An Introduction to Convolutional Neural Networks, arXiv preprint arXiv:1511.08458, 2015. <https://arxiv.org/abs/1511.08458>
30. [KemelmacherShlizerman2016] Kemelmacher-Shlizerman, I., Seitz, S. M., Miller, D., & Brossard, E., The MegaFace Benchmark: 1 Million Faces for Recognition at Scale, Proceedings of CVPR, IEEE, Las Vegas (USA), June 2016, pp. 4873–4882.

10 Anexe

Anexa 1: Codul sursă train.py care antrenează modelul ArcFace sau Softmax

```
"""The training module for ArcFace face recognition."""
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.get_logger().setLevel('ERROR')
from argparse import ArgumentParser
from tensorflow import keras
from dataset import build_dataset
from losses import ArcLoss
from network import ArcLayer, L2Normalization, hrnet_v2
from training_supervisor import TrainingSupervisor

parser = ArgumentParser()
parser.add_argument("--softmax", default=False, type=bool, help="Training with softmax loss.")
parser.add_argument("--epochs", default=60, type=int, help="Number of training epochs.")
parser.add_argument("--batch_size", default=128, type=int, help="Training batch size.")
parser.add_argument("--export_only", default=False, type=bool, help="Save the model without training.")
parser.add_argument("--restore_weights_only", default=False, type=bool, help="Only restore the model weights from checkpoint.")
parser.add_argument("--override", default=False, type=bool, help="Manually override the training objects.")
parser.add_argument("--learning_rate", default=0.001, type=float, help="Initial learning rate.")
parser.add_argument("--use_lr_schedule", default=True, type=bool, help="Use learning rate schedule.")
parser.add_argument("--min_lr", default=0.00001, type=float, help="Minimum learning rate.")
parser.add_argument("--warmup_epochs", default=5, type=int, help="Number of warmup epochs.")
parser.add_argument("--decay_type", default="linear", choices=["linear", "exponential"], help="Decay type after warmup")
parser.add_argument("--skip_validation", default=False, type=bool, help="Skip validation during training.")
parser.add_argument("--val_batch_size", default=None, type=int, help="Batch size for validation.")
args = parser.parse_args()

if __name__ == "__main__":
    name = "hrnetv2_softmax+arcface_intregg_cum_trebuie"
    train_files = "D:\\Licenta\\arcface-main\\faces_emo\\split_dataset\\train.record"
    test_files = "D:\\Licenta\\arcface-main\\faces_emo\\split_dataset\\test.record"
    val_files = "D:\\Licenta\\arcface-main\\faces_emo\\split_dataset\\val.record"
    input_shape = (112, 112, 3)
    embedding_size = 512
    num_ids = 100
```

```

num_examples = 4999
training_dir = os.getcwd()
export_dir = os.path.join(training_dir, 'exported', name)
regularizer = keras.regularizers.L2(5e-4)
frequency = 1000

if args.restore_weights_only:
    checkpoint_path = tf.train.latest_checkpoint(os.path.join(training_dir, 'checkpoints', name))
    if checkpoint_path:
        checkpoint = tf.train.load_checkpoint(checkpoint_path)
        try:
            restored_lr =
checkpoint.get_tensor('schedule/learning_rate/.ATTRIBUTES/VARIABLE_VALUE')
            print(f'Restored LR from checkpoint: {restored_lr}')
            args.learning_rate = float(restored_lr)
        except:
            print("Warning: Could not restore learning rate.")

base_model = hrnet_v2(
    input_shape=input_shape,
    output_size=embedding_size,
    width=18,
    trainable=True,
    kernel_regularizer=regularizer,
    name="embedding_model"
)

if args.softmax:
    print("Building training model with softmax loss...")
    model = keras.Sequential([
        keras.Input(input_shape),
        base_model,
        keras.layers.Dense(num_ids, kernel_regularizer=regularizer),
        keras.layers.Softmax(),
        name="training_model")
    loss_fun = keras.losses.CategoricalCrossentropy()
else:
    print("Building training model with ARC loss...")
    model = keras.Sequential([
        keras.Input(input_shape),
        base_model,
        L2Normalization(),
        ArcLayer(num_ids, regularizer)],
        name="training_model")
    loss_fun = ArcLoss()

model.summary()

optimizer = keras.optimizers.Adam(

```

```

learning_rate=args.learning_rate,
amsgrad=True,
epsilon=0.001,
clipvalue=1.0
)

dataset_train = build_dataset(
    train_files,
    batch_size=args.batch_size,
    one_hot_depth=num_ids,
    training=True,
    buffer_size=4096
)

val_batch_size = args.val_batch_size or args.batch_size
dataset_val = None
if val_files and not args.skip_validation:
    try:
        dataset_val = build_dataset(
            val_files,
            batch_size=val_batch_size,
            one_hot_depth=num_ids,
            training=False,
            buffer_size=4096
        )
        val_iter = iter(dataset_val)
        next(val_iter)
    except Exception as e:
        print(f"Validation error: {e}")
        dataset_val = None

supervisor = TrainingSupervisor(
    model=model,
    optimizer=optimizer,
    loss=loss_fun,
    dataset=dataset_train,
    training_dir=training_dir,
    save_freq=frequency,
    monitor="categorical_accuracy",
    mode='max',
    name=name,
    val_dataset=dataset_val,
    num_examples=num_examples,
    batch_size=args.batch_size,
    epochs=args.epochs,
    warmup_epochs=args.warmup_epochs,
    decay_type=args.decay_type,
    learning_rate=args.learning_rate,
    min_lr=args.min_lr
)

```

```

)
if args.export_only:
    supervisor.restore(args.restore_weights_only, from_scout=True)
    supervisor.export(model, export_dir) # Exportă întregul model
    print(" ✅ Exportat modelul întreg cu capul final din best checkpoint.")
    quit()

supervisor.restore(args.restore_weights_only)

if not args.softmax and args.restore_weights_only:
    with supervisor.clerk.as_default():
        tf.summary.text("⚠️ Faza 2", "Începem faza ArcFace după Softmax",
step=int(supervisor.schedule['step']))
        print("📌 Marker TensorBoard scris: Începem faza ArcFace")

checkpoint_path = tf.train.latest_checkpoint(
    os.path.join(training_dir, 'checkpoints', name))
)
if checkpoint_path:
    checkpoint_reader = tf.train.load_checkpoint(checkpoint_path)
    saved_step =
checkpoint_reader.get_tensor('schedule/step/.ATTRIBUTES/VARIABLE_VALUE')
    saved_epoch =
checkpoint_reader.get_tensor('schedule/epoch/.ATTRIBUTES/VARIABLE_VALUE')
    print(f"Preluăm contoarele din checkpoint: Pas={saved_step}, Epoca={saved_epoch}")
    supervisor.override(step=saved_step, epoch=saved_epoch)

if args.override:
    supervisor.override(0, 0)

steps_per_epoch = num_examples // args.batch_size
supervisor.train(args.epochs, steps_per_epoch)

if args.softmax:
    print("Saving full Softmax model (with classification head)...")
    model.save(export_dir)
else:
    print("Saving only ArcFace backbone (embeddings)...")
    supervisor.export(base_model, export_dir)

```

Anexa 2: Codul sursă training_supervisor.py care asigură supervizarea și checkpointurile

"""This module provides the implementation of training supervisor."""

```

import os
import tensorflow as tf
from tqdm import tqdm

```

```

class TrainingSupervisor(object):

```

```

"""A training supervisor will organize and monitor the training process."""
def __init__(self, model, optimizer, loss, dataset, training_dir, save_freq,
             monitor, mode, name, val_dataset=None, lr_schedule=None, num_examples=None,
             batch_size=None,
             epochs=None, warmup_epochs=5, decay_type='linear', learning_rate=0.001,
             min_lr=1e-5) -> None:
    super().__init__()
    self.model = model
    self.optimizer = optimizer
    self.loss_fun = loss
    self.dataset = dataset
    self.data_generator = iter(self.dataset)
    self.val_dataset = val_dataset
    self.save_freq = save_freq
    self.training_dir = training_dir
    self.name = name
    self.eval_freq = min(save_freq // 5, 200)

    # Metrics setup
    self.metrics = {
        'categorical_accuracy': tf.keras.metrics.CategoricalAccuracy(name='train_accuracy'),
        'loss': tf.keras.metrics.Mean(name="train_loss_mean")
    }
    self.val_metrics = None
    if val_dataset is not None:
        self.val_metrics = {
            'val_categorical_accuracy': tf.keras.metrics.CategoricalAccuracy(name='val_accuracy'),
            'val_loss': tf.keras.metrics.Mean(name='val_loss_mean')
        }

    # Monitoring setup
    self.monitor = self.metrics[monitor]
    self.mode = mode

    # Training state
    self.schedule = {
        'step': tf.Variable(0, trainable=False, dtype=tf.int64),
        'epoch': tf.Variable(0, trainable=False, dtype=tf.int64),
        'monitor_value': tf.Variable(-1.0, trainable=False, dtype=tf.float32),
        'learning_rate': tf.Variable(optimizer.lr.numpy(), trainable=False, dtype=tf.float32)
    }

    # Learning rate schedule setup (optional)
    self.lr_schedule = lr_schedule
    if lr_schedule is None and num_examples is not None and batch_size is not None and
       epochs is not None:
        total_steps = num_examples // batch_size * epochs
        warmup_steps = num_examples // batch_size * warmup_epochs
        total_decay_steps = max(total_steps - warmup_steps, 1)

```

```

def lr_schedule_fn(step):
    step = tf.cast(step, tf.float32)
    if step < warmup_steps:
        return (step / warmup_steps) * learning_rate
    else:
        decay_progress = (step - warmup_steps) / total_decay_steps
        if decay_type == "linear":
            lr = learning_rate * (1 - decay_progress)
        else:
            lr = learning_rate * tf.math.exp(-decay_progress)
        return tf.maximum(lr, min_lr)

    self.lr_schedule = lr_schedule_fn

# Checkpoint setup
self.checkpoint = tf.train.Checkpoint(
    model=self.model,
    optimizer=self.optimizer,
    metrics=self.metrics,
    schedule=self.schedule,
    monitor=self.monitor,
    data_generator=self.data_generator
)
if self.val_metrics is not None:
    self.checkpoint.val_metrics = self.val_metrics

self.manager = tf.train.CheckpointManager(
    self.checkpoint,
    os.path.join(training_dir, 'checkpoints', name),
    max_to_keep=2
)
self.scout = tf.train.CheckpointManager(
    self.checkpoint,
    os.path.join(training_dir, 'model_scout', name),
    max_to_keep=1
)
self.clerk = tf.summary.create_file_writer(
    os.path.join(training_dir, 'logs', name)
)

def restore(self, weights_only=False, from_scout=False):
    checkpoint_path = self.scout.latest_checkpoint if from_scout else
    self.manager.latest_checkpoint
    if not checkpoint_path:
        print("No checkpoint found. Starting from scratch.")
        return
    print(f'Restoring from {"scout' if from_scout else 'regular'} checkpoint: {checkpoint_path}')
    if weights_only:

```

```

        tf.train.Checkpoint(model=self.model).restore(checkpoint_path)
    else:
        status = self.checkpoint.restore(checkpoint_path)
        status.expect_partial()
        self.optimizer.lr.assign(self.schedule['learning_rate'].numpy())
        print(f'Restored learning rate: {self.optimizer.lr.numpy()}')

@tf.function
def _train_step(self, x_batch, y_batch, step):
    if self.lr_schedule is not None:
        new_lr = self.lr_schedule(step)
        self.optimizer.lr.assign(new_lr)
        self.schedule['learning_rate'].assign(new_lr)

    with tf.GradientTape() as tape:
        logits = self.model(x_batch, training=True)
        loss = self.loss_fun(y_batch, logits) + sum(self.model.losses)

        grads = tape.gradient(loss, self.model.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_weights))
    return logits, loss

@tf.function
def _update_metrics(self, labels, logits, loss):
    self.metrics['categorical_accuracy'].update_state(labels, logits)
    self.metrics['loss'].update_state(loss)

@tf.function
def _evaluate_step(self, x, y):
    logits = self.model(x, training=False)
    loss = self.loss_fun(y, logits) + sum(self.model.losses)
    self.val_metrics['val_categorical_accuracy'].update_state(y, logits)
    self.val_metrics['val_loss'].update_state(loss)
    return loss

def _reset_metrics(self, reset_val=True):
    for metric in self.metrics.values():
        metric.reset_states()
    if reset_val and self.val_metrics is not None:
        for metric in self.val_metrics.values():
            metric.reset_states()

def _evaluate_model(self):
    if self.val_dataset is None:
        return None
    print(f'\n[Epoch {int(self.schedule['epoch']) + 1}] Starting validation...')
    self._reset_metrics(reset_val=True)
    for x_batch, y_batch in iter(self.val_dataset):
        self._evaluate_step(x_batch, y_batch)

```

```

val_loss = self.val_metrics['val_loss'].result()
val_acc = self.val_metrics['val_categorical_accuracy'].result()
print(f"[Validation] Accuracy: {val_acc:.4f}, Loss: {val_loss:.2f}")
return (val_loss, val_acc)

def _log_to_tensorboard(self, evaluate=True):
    current_step = int(self.schedule['step'])
    train_loss = self.metrics['loss'].result()
    train_acc = self.metrics['categorical_accuracy'].result()
    lr = self.optimizer.lr.numpy()

    val_results = None
    if evaluate and self.val_metrics is not None:
        val_results = self._evaluate_model()

    with self.clerk.as_default():
        tf.summary.scalar("training/loss", train_loss, step=current_step)
        tf.summary.scalar("training/accuracy", train_acc, step=current_step)
        tf.summary.scalar("training/learning_rate", lr, step=current_step)
        if val_results:
            val_loss, val_acc = val_results
            tf.summary.scalar("validation/loss", val_loss, step=current_step)
            tf.summary.scalar("validation/accuracy", val_acc, step=current_step)

    print(f"\n[Step {current_step}]")
    print(f"LR: {lr:.7f} | Train Acc: {train_acc:.4f} | Train Loss: {train_loss:.2f}")
    if val_results:
        print(f"Val Acc: {val_acc:.4f} | Val Loss: {val_loss:.2f}")

def _checkpoint(self):
    current = self.val_metrics['val_categorical_accuracy'].result()
    previous = self.schedule['monitor_value'].numpy()

    if previous < 0.0 or (current > previous and self.mode == 'max') or (current < previous and self.mode == 'min'):
        print(f"Monitor improved ({previous:.4f} → {current:.4f}). Saving best checkpoint...")
        self.schedule['monitor_value'].assign(current)
        self.scout.save()

    self.manager.save()
    print(f'Checkpoint saved at step {int(self.schedule['step'])}')

def _save_model(self, export_dir):
    try:
        self.model.save(export_dir, save_format='tf')
    except Exception as e:
        print(f"Failed to save full model to {export_dir}: {e}")

def train(self, epochs, steps_per_epoch):

```

```

initial_epoch = int(self.schedule['epoch'].numpy())
total_epochs = initial_epoch + epochs

print(f"\n==== Training Summary ===")
print(f"Starting from epoch: {initial_epoch + 1}")
print(f"Training for {epochs} epochs (total: {total_epochs} epochs)")
print(f"Steps per epoch: {steps_per_epoch}")
print(f"Validation: {'enabled' if self.val_dataset else 'disabled'}\n")

for epoch in range(initial_epoch, total_epochs):
    print(f"\n==== Epoch {epoch + 1}/{total_epochs} ===")
    progress_bar = tqdm(total=steps_per_epoch,
                         bar_format='{l_bar}{bar}| {n_fmt}/{total_fmt} [{elapsed}<{remaining}]',
                         colour='green',
                         dynamic_ncols=True,
                         leave=False)

    for step, (x_batch, y_batch) in enumerate(self.data_generator):
        logits, loss = self._train_step(x_batch, y_batch, self.schedule['step'])
        self._update_metrics(y_batch, logits, loss)
        self.schedule['step'].assign_add(1)

        progress_bar.update(1)
        progress_bar.set_postfix({
            "loss": f"{loss.numpy():.2f}",
            "acc": f"{self.metrics['categorical_accuracy'].result().numpy():.3f}",
            "lr": f"{self.optimizer.lr.numpy():.6f}"
        })

    if (step + 1) % self.eval_freq == 0:
        self._log_to_tensorboard()
        self._checkpoint()
    if (step + 1) % self.save_freq == 0:
        self._checkpoint()

    self.schedule['epoch'].assign_add(1)
    self.data_generator = iter(self.dataset)
    self._log_to_tensorboard()
    self._checkpoint()
    progress_bar.close()

print("\n==== Training Completed ===")

def export(self, model, export_dir):
    print(f"\nExporting model to {export_dir}...")
    model.save(export_dir, save_format='tf')
    print("Model exported successfully!")

def override(self, step=None, epoch=None, monitor_value=None):

```

```

if step is not None:
    self.schedule['step'].assign(step)
if epoch is not None:
    self.schedule['epoch'].assign(epoch)
if monitor_value is not None:
    self.schedule['monitor_value'].assign(monitor_value)

```

Anexa 3: Codul sursă evaluate.py care evaluează performanța modelului

```

import argparse
import random
import os

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf
from sklearn.metrics import (ConfusionMatrixDisplay, auc, confusion_matrix,
                             roc_curve)
from sklearn.metrics.pairwise import cosine_similarity
from tqdm import tqdm

from dataset import build_dataset
from losses import ArcLoss
from network import ArcLayer, L2Normalization

# ===== Helper utilities =====

def _to_uint8(img, eps=1e-8):
    """
    Convertește orice tensor imagine în uint8 afișabil.
    - Dacă e deja uint8 → return.
    - Altfel rescalează (min-max) la [0,255].
    """
    if img.dtype == np.uint8:
        return img

    img = np.asarray(img, dtype=np.float32)
    v_min, v_max = img.min(), img.max()
    if v_max - v_min < eps:    # imagine constantă
        return np.zeros_like(img, dtype=np.uint8)

    img = (img - v_min) / (v_max - v_min)  # [0,1]
    img = (img * 255.0).clip(0, 255).astype(np.uint8)
    return img

def _save_debug(img, fname_np="debug.npy", fname_hist="debug_hist.png"):
    """
    Salvează tensor brut + histogramă valorilor.
    """

```

```

np.save(fname_np, img)
plt.figure()
flat = img.flatten()
plt.hist(flat, bins=50, color="gray")
plt.title("Histogramă valori brute")
plt.xlabel("Valoare pixel"); plt.ylabel("Frecvență")
plt.tight_layout(); plt.savefig(fname_hist); plt.close()
print(f"🔍 Tensor brut salvat în '{fname_np}' + histogramă în '{fname_hist}'.")

```



```

def _plot_confusion_examples(imgs, y_true, y_pred,
                            max_examples=3, prefix="confusion_example"):
    """Exemple de clasificări greșite (confuzii)."""
    mis = [i for i, (t, p) in enumerate(zip(y_true, y_pred)) if t != p]
    if not mis:
        print("✅ Nicio confuzie.")
        return

    shown = 0
    for idx in mis:
        if shown >= max_examples:
            break
        true_lbl, pred_lbl = y_true[idx], y_pred[idx]
        try:
            alt_idx = next(j for j, lbl in enumerate(y_true) if lbl == pred_lbl and j != idx)
        except StopIteration:
            alt_idx = idx

        # SALVĂM tensorul brut al primului exemplu confuz pentru debug
        if shown == 0:
            _save_debug(imgs[idx])

        img_a = _to_uint8(imgs[idx])
        img_b = _to_uint8(imgs[alt_idx])

        fig, axes = plt.subplots(1, 2, figsize=(6, 3))
        axes[0].imshow(img_a); axes[0].set_title(f"Corect: {true_lbl}"); axes[0].axis("off")
        axes[1].imshow(img_b); axes[1].set_title(f"Prezicere: {pred_lbl}"); axes[1].axis("off")
        plt.suptitle(f"Confuzie #{shown + 1}")
        plt.tight_layout()
        fname = f"{prefix}_{shown + 1}.png"
        plt.savefig(fname); plt.show()
        print(f"🖼️ Confuzie salvată în '{fname}'.")

        shown += 1

```



```

def _plot_correct_examples(imgs, y_true, y_pred,
                           max_examples=3, prefix="correct_example"):
    """Exemple corecte: două imagini diferite din aceeași clasă."""

```

```

correct = [i for i, (t, p) in enumerate(zip(y_true, y_pred)) if t == p]
if len(correct) < 2:
    print("⚠ Prea puține corecte pentru vizualizare.")
    return

shown, used = 0, set()
for idx in correct:
    if shown >= max_examples or idx in used:
        continue
    label = y_true[idx]
    try:
        alt_idx = next(j for j in correct
                       if j != idx and j not in used and y_true[j] == label)
    except StopIteration:
        continue

    # SALVĂM tensor brut al primului exemplu corect (dacă încă nu l-am salvat)
    if not os.path.exists("debug.npy"):  # dacă nu s-a salvat la confuzie
        _save_debug(imgs[idx])

    img_a = _to_uint8(imgs[idx])
    img_b = _to_uint8(imgs[alt_idx])

    fig, axes = plt.subplots(1, 2, figsize=(6, 3))
    axes[0].imshow(img_a); axes[0].set_title(f"Corect: {label}"); axes[0].axis("off")
    axes[1].imshow(img_b); axes[1].set_title(f"Alt exemplu: {label}"); axes[1].axis("off")
    plt.suptitle(f"Predictie corecta #{shown + 1}")
    plt.tight_layout()
    fname = f"{prefix}_{shown + 1}.png"
    plt.savefig(fname); plt.show()
    print(f"✓ Exemplu corect salvat în '{fname}'")
    used.update([idx, alt_idx])
    shown += 1


def _plot_top_confusions_heatmap(cm, top_n=20, fname="heatmap_top_confusions.png"):
    if cm.size == 0:
        return
    errors = cm.copy().astype(float); np.fill_diagonal(errors, 0)
    tot = errors.sum(1) + errors.sum(0)
    if (tot == 0).all(): return
    idx = np.argsort(tot)[::-1][:min(top_n, (tot > 0).sum())]
    sub = cm[np.ix_(idx, idx)].astype(float)
    sub = sub / sub.sum(1, keepdims=True)
    plt.figure(figsize=(1+0.4*len(idx), 0.8+0.4*len(idx)))
    sns.heatmap(sub, annot=True, fmt=".2f", cmap="Blues",
                xticklabels=idx, yticklabels=idx)
    plt.title(f"Top {len(idx)} clase confundate (normalizat pe rând)")
    plt.xlabel("Predicted"); plt.ylabel("True")

```

```

plt.tight_layout(); plt.savefig(fname); plt.show()
print(f"🔥 Heatmap în '{fname}'.")

# ===== Model loading =====

def load_model_with_custom_objects(path):
    print(f"⚙️ Încarcăm modelul din: {path}")
    return tf.keras.models.load_model(
        path,
        custom_objects={"ArcLoss": ArcLoss,
                        "ArcLayer": ArcLayer,
                        "L2Normalization": L2Normalization}
    )

# ===== Evaluations =====

def evaluate_classification(model, ds, use_softmax):
    if use_softmax:
        model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
                      metrics=[tf.keras.metrics.CategoricalAccuracy()])
    else:
        model.compile(loss=ArcLoss(),
                      metrics=[tf.keras.metrics.CategoricalAccuracy()])
    loss, acc = model.evaluate(ds); print(f"\n🎯 Acc={acc:.4f} | Loss={loss:.4f}")

    y_true, y_pred, imgs = [], [], []
    for xb, yb in ds:
        p = model.predict(xb, verbose=0)
        y_true.extend(np.argmax(yb.numpy(), 1))
        y_pred.extend(np.argmax(p, 1))
        imgs.extend(xb.numpy())
    y_true, y_pred = np.array(y_true), np.array(y_pred)

    cm = confusion_matrix(y_true, y_pred)
    ConfusionMatrixDisplay(cm).plot(cmap=plt.cm.Blues, xticks_rotation="vertical")
    plt.title("Matrice de confuzie – toate clasele")
    plt.tight_layout(); plt.savefig("confusion_matrix.png"); plt.show()

    _plot_top_confusions_heatmap(cm)
    _plot_confusion_examples(imgs, y_true, y_pred)
    _plot_correct_examples(imgs, y_true, y_pred)

# ----- ArcFace similarity evaluation ----->

def generate_pairs(emb, lbl, n=1000):
    lbl2idx = {}
    for i, l in enumerate(lbl): lbl2idx.setdefault(l, []).append(i)
    pairs, y = [], []

```

```

for _ in range(n//2):          # pozitive
    l = random.choice([l for l in lbl2idx if len(lbl2idx[l])>1])
    i1, i2 = random.sample(lbl2idx[l], 2)
    pairs.append((emb[i1], emb[i2])); y.append(1)
L = list(lbl2idx.keys())
for _ in range(n//2):          # negative
    i1, i2 = random.sample(L, 2)
    i1 = random.choice(lbl2idx[i1]); i2 = random.choice(lbl2idx[i2])
    pairs.append((emb[i1], emb[i2])); y.append(0)
return pairs, y

def evaluate_arcface_similarity(model, ds):
    emb, lbl = [], []
    for xb, yb in tqdm(ds, desc="🕒 embeddings"):
        emb.append(model(xb).numpy())
        lbl.append(np.argmax(yb.numpy(), 1))
    emb, lbl = np.concatenate(emb), np.concatenate(lbl)
    pairs, y = generate_pairs(emb, lbl)
    sims = [cosine_similarity([a], [b])[0,0] for a,b in pairs]
    fpr, tpr, _ = roc_curve(y, sims); auc_val = auc(fpr, tpr)
    print(f"\n🕒 ROC AUC={auc_val:.4f}")
    plt.figure(); plt.plot(fpr, tpr, lw=2, label=f"AUC={auc_val:.4f}")
    plt.plot([0,1],[0,1], lw=2, ls="--"); plt.legend(); plt.grid()
    plt.xlabel("FPR"); plt.ylabel("TPR"); plt.tight_layout()
    plt.savefig("roc_curve.png"); plt.show()
    # histogram
    plt.figure()
    plt.hist([s for s,l in zip(sims,y) if l==1], bins=30, alpha=.6, label="Pozitive")
    plt.hist([s for s,l in zip(sims,y) if l==0], bins=30, alpha=.6, label="Negative")
    plt.legend(); plt.xlabel("Cosine"); plt.ylabel("Freq"); plt.tight_layout()
    plt.savefig("similaritate_cosine.png"); plt.show()

# ===== Main
=====

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--model_path", required=True)
    ap.add_argument("--test_record",
                    default="D:\\Licenta\\arcface-main\\faces_emo\\split_dataset\\test.record")
    ap.add_argument("--num_ids", type=int, default=100)
    ap.add_argument("--batch_size", type=int, default=8)
    ap.add_argument("--softmax", action="store_true")
    args = ap.parse_args()

    ds = build_dataset(args.test_record, batch_size=args.batch_size,
                       one_hot_depth=args.num_ids, training=False, buffer_size=4096)
    model = load_model_with_custom_objects(args.model_path)
    evaluate_classification(model, ds, args.softmax)

```

```

if not args.softmax:
    evaluate_arcface_similarity(model, ds)

if __name__ == "__main__":
    main()

```

Anexa 4: Codul sursă losses.py care definește funcția de pierdere ArcLoss

```

import tensorflow as tf
from math import pi

class ArcLoss(tf.keras.losses.Loss):
    def __init__(self, margin=0.5, scale=64, name="arcloss"):
        super().__init__(name=name)
        self.margin = margin
        self.scale = scale
        self.threshold = tf.math.cos(pi - margin)
        self.cos_m = tf.math.cos(margin)
        self.sin_m = tf.math.sin(margin)
        self.safe_margin = tf.constant(0.2, dtype=tf.float32) # Valoare fixă pentru stabilitate

    @tf.function
    def call(self, y_true, y_pred):
        cos_t = y_pred
        sin_t = tf.math.sqrt(1 - tf.math.square(cos_t))
        cos_t_margin = tf.where(
            cos_t > self.threshold,
            cos_t * self.cos_m - sin_t * self.sin_m,
            cos_t - self.safe_margin
        )

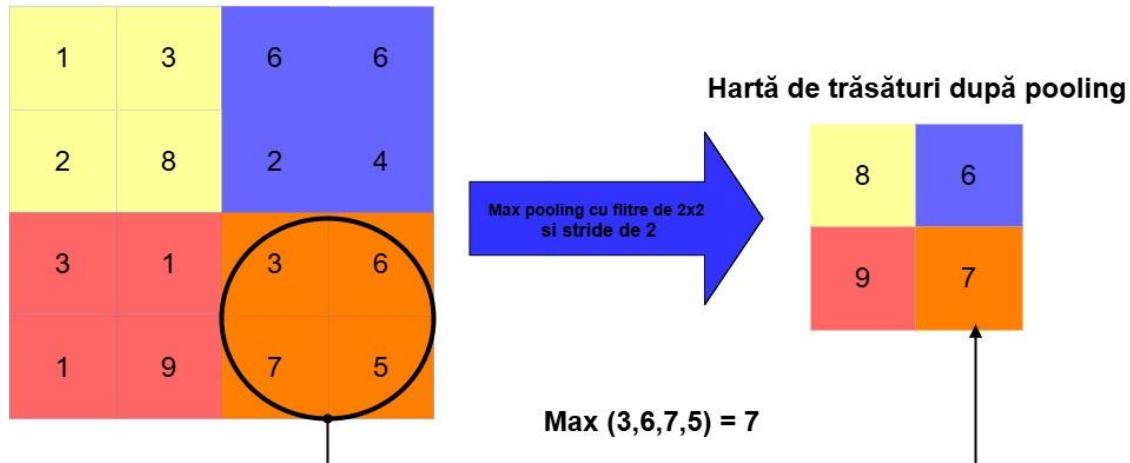
        mask = y_true
        cos_t_onehot = cos_t * mask
        cos_t_margin_onehot = cos_t_margin * mask
        logits = (cos_t + cos_t_margin_onehot - cos_t_onehot) * self.scale
        return tf.nn.softmax_cross_entropy_with_logits(y_true, logits)

    def get_config(self):
        config = super().get_config()
        config.update({"margin": self.margin, "scale": self.scale})
        return config

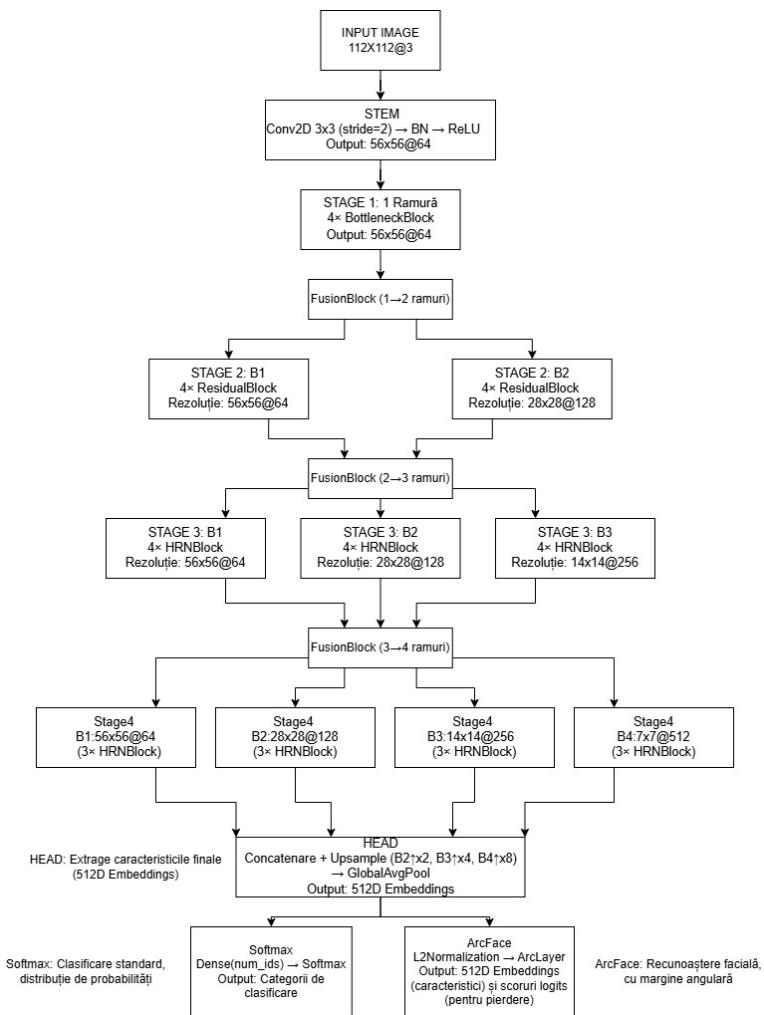
```

Anexa 5: Figura cu Exemplu de operație Max Pooling cu filtru 2×2 și pas (stride) 2 aplicată pe o hartă de trăsături.

Hartă de trăsături rectificată



Anexa 6: Figura cu Arhitectura completă a modelului propus, cu HRNet și cap de rețea Softmax / ArcFace.





Vlad-Alexandru MAIE R

Date of birth: 13/10/2002

Nationality: Romanian

CONTACT

Strada Garoafei, Nr. 2, Sc. C,
Ap. 26
420153 Bistrița, Romania
(Home)

vladmaier13@gmail.com

(+40) 757651805



europass

EDUCATION AND TRAINING

03/10/2021 – CURRENT Cluj-Napoca, Romania

Bachelor's Degree (Ongoing) Technical University of Cluj-Napoca

Address Strada Memorandumului , Nr. 28, 400114, Cluj-Napoca, Romania |
Website <https://www.utcluj.ro> | **Field of study** Electronics, Telecommunications and Information Technology

01/09/2017 – 01/07/2021 Bistrița, Romania

Baccalaureate Diploma Technical College "INFOEL" Bistrita

Address Calea Moldovei, Nr. 20, 410444, Bistrița, Romania | **Website** <https://infoel.ro> | **Field of study** Mathematics and Informatics (Intensive Informatics) | **Final grade** 8,03

01/09/2017 – 12/07/2021 Bistrița, Romania

Professional competence certificate Technical College "INFOEL" Bistrita

1. Development of the design and structure of the necessary software products for implementing software systems, software applications, databases, and web pages (client-oriented software).
2. Customization, configuration, and modification of software applications to adapt to the client's information systems.

Address Calea Moldovei, Nr. 20, 410444, Bistrița, Romania | **Website** <https://infoel.ro> | **Field of study** Mathematics and Informatics (Intensive Informatics)

01/09/2017 – 12/07/2021 Bistrița, Romania

Digital Skills Certificate (Experienced User Level) Technical College "INFOEL" Bistrita

Address Calea Moldovei , Nr. 20, 410444, Bistrița, Romania | **Website** <https://infoel.ro>

01/09/2017 – 12/07/2021 Bistrița, Romania

Certificate of Language Proficiency Technical College "INFOEL" Bistrita

Address Calea Moldovei, Nr. 20, 410444, Bistrița, Romania | **Website** <https://infoel.ro> | **Field of study** ENGLISH | **Final grade** B1

ABOUT ME

Description:

I am a recent graduate of the Faculty of Electronics, Telecommunications and Information Technology at the Technical University of Cluj-Napoca, currently preparing to present my bachelor's thesis. I am a fast learner, adaptable, and driven by integrity and perseverance. I thrive in environments that offer feedback and growth, and I am now looking forward to applying and expanding my skills in engineering and applied programming.

LANGUAGE SKILLS

MOTHER TONGUE(S): Romanian

Other language(s):

English

Listening C1

Spoken production B1

Reading C1

Spoken interaction B2

Writing B1

Levels: A1 and A2: Basic user; B1 and B2: Independent user; C1 and C2: Proficient user

SKILLS

Microsoft Office | HTML | Outlook | Internet Navigation | OrCAD (Beginner) | SQL and MS-SQL | Social networks/
Social Media | MATLAB | LTSpice | Circuit Simulation | Beginner JavaScript for Web Development | C & C++ | Java | Python
Proteus (Beginner) | Google Drive | Vivado | Microcontroller Programming | ASSEMBLY 8086 | Python (computer
programming) | Unity (digital game creation systems) | Spyder - Anaconda | visual studio, visual Basic

DRIVING LICENCE

- **Driving Licence:** AM
- **Driving Licence:** B1
- **Driving Licence:** B

PROJECTS

- 10/2023 – 01/2024**
- **USB charger from Li-Ion 3.7V battery + protection (Boost source)**
This University project was about implementing a DC-DC converter circuit by means of a lithium-ion battery to convert its power and supply some customer device via a USB port. The input DC voltage was 3.75 V and the output DC voltage was to be regulated at 5 V. The boost design with a switch was employed for this purpose. Switching power supplies are more efficient than linear ones. They provide switching, step down, step up, and inversion. Furthermore, certain systems can be designed to create an isolation of the output voltage from the input. This project was implemented and simulated in both LTSpice and Proteus.

10/2023 – 01/2024

 - **Dynamic signaling system designed for vehicles**
This university project focused on developing a dynamic signaling system for vehicles. The project involved designing a circuit in VIVADO to manage both regular signaling and hazard lights. Furthermore, it required the practical deployment of this system on a Basys 3 board. Through programming, I configured specific pins and LEDs on the board to facilitate various signaling modes, all of which could be switched using onboard buttons

02/2023 – 01/2024

 - **Circuit for controlling pressure in a hyperbaric oxygen chamber**
This university project consisted in projecting the circuit for monitoring and controlling the level of pressure inside a hyperbaric chamber. The pressure was maintained in a specified range with the help of a pump, controlled by a comparator and an electromagnetic relay. The pump-relay assembly was modeled with a resistor. The state of the pump (on/off) was signaled by a LED with a specified color. The project was only implemented and simulated in OrCAD.

11/2024 – 01/2025

 - **Smart Parking Management System with Cloud Monitoring and SMS Notifications**
I developed an intelligent parking management system using Python, AWS cloud services, and Twilio for SMS notifications. The solution monitored available parking spots, occupancy duration, and waiting times, storing

data in JSON format and displaying it through intuitive gauges and line charts. It delivered real-time updates and sent personalized alerts to users, helping to improve parking space efficiency and enhance user experience.

11/2024 – 01/2025

Simulation of OFDM Transmission with Multipath Channel and Noise

In this academic project, I simulated a communication system based on Orthogonal Frequency Division Multiplexing (OFDM) using 16-QAM modulation. The work involved generating symbols, applying the IFFT, adding a cyclic prefix, and modeling both multipath channels and AWGN noise, followed by equalizing the received signal. I evaluated system performance by calculating the bit error rate (BER) under various propagation scenarios, highlighting the robustness of OFDM techniques against interference and signal distortions.

01/2024 – 04/2025

Mini-game developed in Unity

I designed and implemented a simple 3D mini-game in Unity as part of a university project. The game challenged players to control a character that collects objects while avoiding obstacles within a defined level. I developed the gameplay mechanics in C#, created basic animations, and managed collision detection and scoring. The project was thoroughly tested and iterated to ensure correct game logic and deliver a smooth user experience.

02/2025 – 04/2025

Shape Reconstruction with Active Shape Models (ASM)

I developed a shape reconstruction system using Active Shape Models (ASM) in Python. The project involved pre-processing images, aligning landmark points with an extended Procrustes analysis, and building a statistical shape model through PCA. I integrated an automated detection step for initialization and implemented iterative fitting of the model to new images. The solution was tested on facial images from the 300W dataset, achieving stable shape adaptation even under scale and translation variations. This project demonstrated a robust approach to modeling and segmenting deformable objects, with potential applications in medical image analysis and facial recognition.

10/2024 – 07/2025

Modern Facial Recognition System with HRNet and ArcFace

I developed a facial recognition system combining the HRNet architecture with advanced loss functions, including ArcFace, to generate robust facial embeddings for reliable identity classification. The system was implemented in Python with TensorFlow, using a modular structure to allow flexible training and evaluation. I trained and tested the models on a cleaned subset of the MS1M-ArcFace dataset, comparing three scenarios (Softmax, ArcFace, and a hybrid approach) with consistent and interpretable results. The evaluation included accuracy, confusion matrices, cosine similarity histograms, and ROC curves, confirming excellent discrimination capabilities (AUC over 0.99). This project provides a solid foundation for scalable and practical facial recognition systems in real-world conditions.

HOBBIES AND INTERESTS

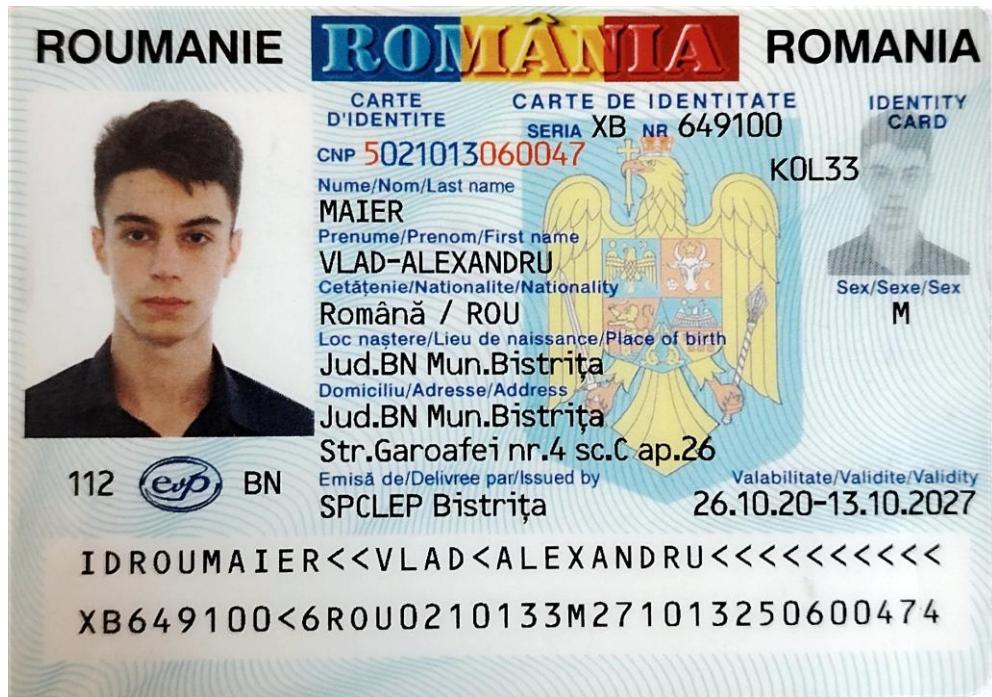
Hobbies:

Volleyball, Cooking, Hiking, Reading, Traveling, Gaming, Going to the Gym

COMMUNICATION AND INTERPERSONAL SKILLS

Abilities:

- **Collaboration and Teamworking**
- **Good Listening Abilities**
- **Problem-solving**
- **Conflict Management Resolution**
- **Personal and Interpersonal Awareness**
- **Emotional Intelligence**



Conform cu originalul