Documentație proiect ML

Lepădatu Teodor

1 Introducere

Acest document descrie procesul de dezvoltare a unui model de *machine learning* care poate clasifica imagini deepfake în 5 categorii diferite, cu o acuratețe cât mai mare. Mai departe voi prezenta încercările pe care le-am făcut timp de două săptămâni.

2 SVM

Fișier `SVM.py`

2.1 Încărcarea datelor

Funcția `load_flattened_images`:

- Pentru fiecare image_id, imaginea `.png` este încărcată și convertită în spațiul RGB.
- Imaginea este transformată într-un vector $\mathbf{x} \in \mathbb{R}^d$ prin metoda `ravel()`, unde $d = H \times W \times 3$.
- Datele sunt returnate ca matrice $X \in \mathbb{R}^{n \times d}$ și, dacă există, vectorul etichetelor $y \in \{0, \ldots, K-1\}^n$.

2.2 Normalizare (*StandardScaler*)

StandardScaler efectuează transformarea

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j},$$

unde μ_j și σ_j sunt media și abaterea standard ale caracteristicii j pe setul de antrenare. Această etapă este importantă, deoarece PCA și SVM sunt sensibile la scara caracteristicilor.

2.3 Reducerea dimensionalității prin PCA

2.3.1 Formulă matematică

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^\top, \quad \Sigma\mathbf{v}_k = \lambda_k\mathbf{v}_k,$$

unde Σ este matricea de covarianță, iar $(\lambda_k, \mathbf{v}_k)$ sunt valorile și vectorii proprii.

2.3.2 Proiecție

Dacă reținem primele $K = 200$ componente:

$$\mathbf{z}_i = V_K^\top(\mathbf{x}_i - \bar{\mathbf{x}}),$$

unde $V_K = [\,\mathbf{v}_1, \ldots, \mathbf{v}_{200}\,]$. Alegerea lui $K = 200$ se bazează pe un compromis între:

- reținerea variației semnificative a datelor,
- reducerea zgomotului și a timpului de calcul,
- evitarea supraînvățării.

2.4 Clasificare SVM cu kernel RBF

2.4.1 Formularea optimizării

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{s.t.} \quad y_i\big(\langle\mathbf{w}, \phi(\mathbf{z}_i)\rangle + b\big) \geq 1 - \xi_i,\ \xi_i \geq 0.$$

2.4.2 Kernel RBF

$$K(\mathbf{z}, \mathbf{z}') = \exp\big(-\gamma\|\mathbf{z} - \mathbf{z}'\|^2\big).$$

Am ales $\gamma = \text{scale}$ (echivalent cu $\gamma = 1/(\text{n\_features} \cdot \text{Var}(X))$) și $C = 1.0$ ca valori implicite, pentru un echilibru între lățimea marginii și penalizarea erorilor. `random_state=42` asigură reproductibilitatea rezultatelor.

2.5 Evaluare

După antrenare, modelul a fost evaluat pe setul de validare, obținând

$$\text{acuratețe} = 63\%.$$

Pe setul de test, s-a obținut o acuratețe de 62.36%.

3 KNN

Fișier `KNN.py`

3.1 Încărcarea datelor

Funcția `load_dataset`:

- Încarcă fișierul `.csv` cu coloana image_id și, dacă există, coloana label.
- Pentru fiecare image_id:
  - Imaginea `.png` este încărcată, convertită în RGB și redimensionată la 100×100 pixeli.
  - Pixelii sunt normalizați în intervalul $[0, 1]$ prin împărțirea la 255.
  - Imaginea este flattenată într-un vector $\mathbf{x} \in \mathbb{R}^d$, cu $d = 100 \times 100 \times 3$.
- Datele sunt returnate ca matrice $X \in \mathbb{R}^{n \times d}$ și, dacă există, vectorul etichetelor $y \in \{0, \ldots, K-1\}^n$.

3.2 Preprocesare

- **Scalare (*StandardScaler*)**:
$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j},$$
unde μ_j, σ_j sunt media și abaterea standard a caracteristicii j.
- **Reducere dimensionalitate (PCA)**:
$$\mathbf{z}_i = V_{200}^\top (\mathbf{x}_i - \bar{\mathbf{x}}),$$
cu $V_{200} \in \mathbb{R}^{d \times 200}$ matricea vectorilor proprii corespunzători celor mai mari 200 valori proprii.

3.3 Clasificator KNN

Fie $\mathbf{z}$ un punct nou de clasificat. Definim mai întâi mulțimea celor K vecini prin:
$$\mathcal{N}_K(\mathbf{z}) = \underset{I \subseteq \{1,\ldots,n\},\, |I|=K}{\arg\min} \sum_{i \in I} \|\mathbf{z} - \mathbf{z}_i\|,$$
unde $\|\mathbf{z} - \mathbf{z}_i\|$ este distanța euclidiană în spațiul proiectat. Eticheta prezisă este:
$$\hat{y}(\mathbf{z}) = \text{mode}\big(\{y_i : i \in \mathcal{N}_K(\mathbf{z})\}\big).$$

3.4 Evaluare

- Modelul este evaluat pe setul de validare.
- `accuracy_score` calculează proporția de clasificări corecte:

$$\text{acuratețe} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{y_i = \hat{y}_i\}.$$

- Cea mai bună acuratețe a fost atinsă pentru $K = 2$: 36.28%.
- Graficul performanței în funcție de K:

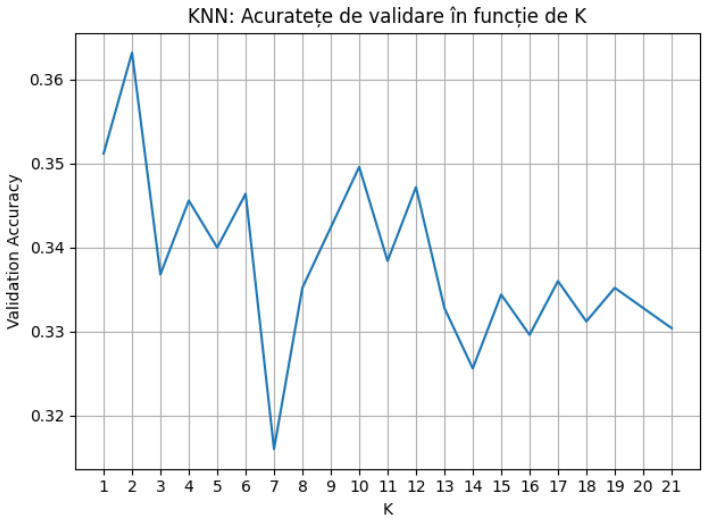

Figure 1: Acuratețe de validare în funcție de K pentru clasificatorul KNN

4 Naive Bayes

Fișier `NaiveBayes.py`

4.1 Încărcarea datelor

Funcția `load_flat_images`:

- Pentru fiecare image_id:
  - Imaginea `.png` este încărcată și convertită în RGB.

  - Imaginea este flattenată într-un vector $\mathbf{x} \in \mathbb{R}^d$, unde $d = H \times W \times 3$.

- Se returnează matricea $X \in \mathbb{R}^{n \times d}$ și, dacă există, vectorul etichetelor $y \in \{0, \ldots, K-1\}^n$.

4.2 Scalare și reducere dimensionalitate

- **StandardScaler**:
$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j},$$
unde μ_j și σ_j sunt media și abaterea standard a caracteristicii j.

- **PCA**:
$$\mathbf{z}_i = V_{200}^\top (\mathbf{x}_i - \bar{\mathbf{x}}),$$
unde V_{200} conține cei mai importanți 200 vectori proprii ai matricei de covarianță.

4.3 Clasificator Gaussian Naive Bayes

Probabilitatea ca o probă $\mathbf{z}$ să aparțină clasei c:

$$P(c \mid \mathbf{z}) = \frac{P(\mathbf{z} \mid c)\, P(c)}{P(\mathbf{z})}.$$

Sub ipoteza independenței caracteristicilor și cu densitate gaussiană:

$$P(z_j \mid c) = \frac{1}{\sqrt{2\pi\, \sigma_{c,j}^2}} \exp\Big(-\frac{(z_j - \mu_{c,j})^2}{2\sigma_{c,j}^2}\Big),$$

iar decizia finală:

$$\hat{y}(\mathbf{z}) = \arg\max_c \Big(\log P(c) \; + \; \sum_{j=1}^{d} \log P(z_j \mid c)\Big).$$

4.4 Evaluare

- Evaluare pe setul de validare cu `accuracy_score`:
$$\text{acuratețe} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{y_i = \hat{y}_i\}.$$

- Acuratețea obținută: 45.86%.

5 Rețea neuronală MLP

Fișier `MLP.py`

5.1 Încărcarea datelor

Funcția `load_flattened_images`:

- Pentru fiecare image_id:
  - Imaginea `.png` este încărcată și convertită în RGB.
  - Imaginea este flattenată într-un vector $\mathbf{x} \in \mathbb{R}^d$, unde $d = H \times W \times 3$.
- Se returnează matricea $X \in \mathbb{R}^{n \times d}$ și, dacă există, vectorul etichetelor $y \in \{0, \ldots, K-1\}^n$.

5.2 Pipeline și arhitectură MLP

- **StandardScaler**: datele sunt standardizate pentru a avea media zero și deviația standard unitară.
- **MLPClassifier**: rețea neuronală cu două straturi ascunse de dimensiuni 256 și 512, activare *ReLU*, *solver* Adam și learning_rate_init = 10^{-3}.

5.3 Antrenare și evaluare

- Modelul este antrenat în 100 de epoci și `batch_size`=64.
- După antrenare, acuratețea pe setul de validare este 57%.
- Pe datele de test, acuratețea este 56.14%.

5.4 Evoluția erorii de antrenament

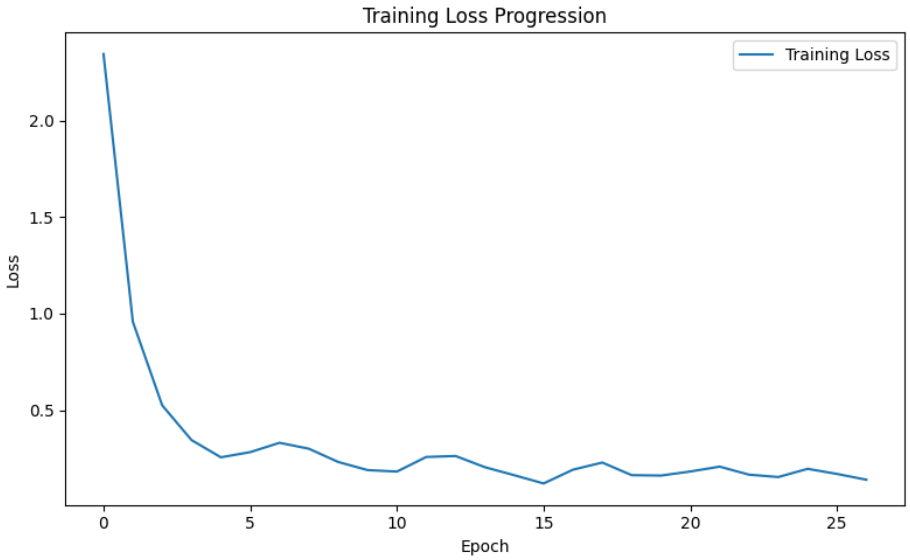

Figure 2: Evoluție loss pe epoci

6 Rețele neuronale convoluționale (CNN)

Am ales acest tip de rețea neuronală pentru că este specializată pe clasificarea imaginilor. Pentru eficientizare și folosirea GPU am folosit *PyTorch*.

6.1 CNN echivalent cu MLP

Fișier `CNN.py`

Prima dată am folosit un CNN cu aceeași structură de straturi complet conectate (256, 512). Am antrenat 10 epoci și am reținut cel mai bun model după acuratețea pe validare. Nu am aplicat augmentare în această etapă.

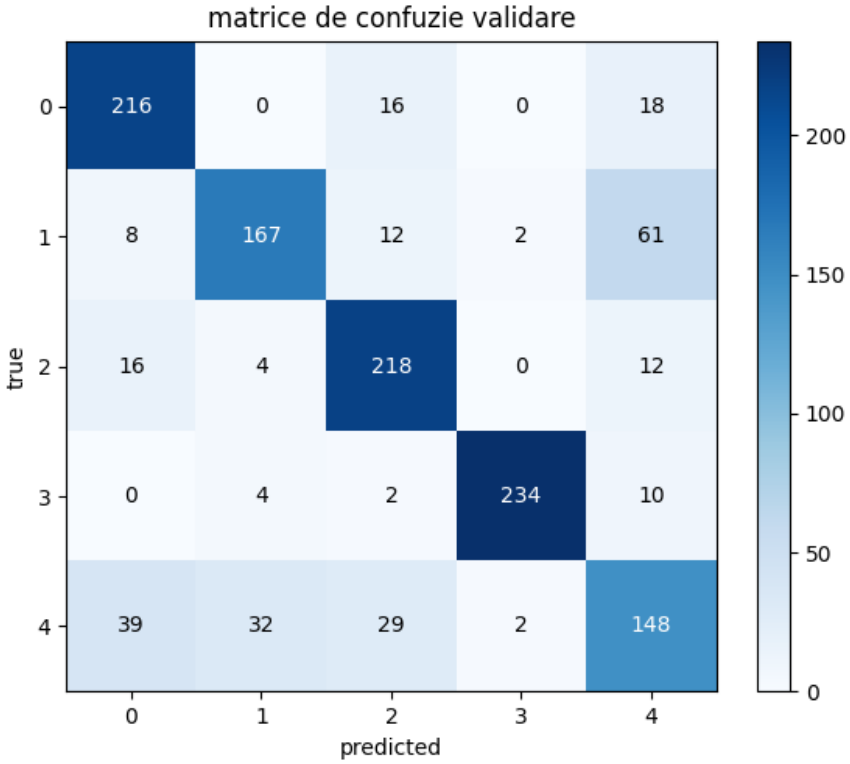

Figure 3: Matricea de confuzie pe setul de validare

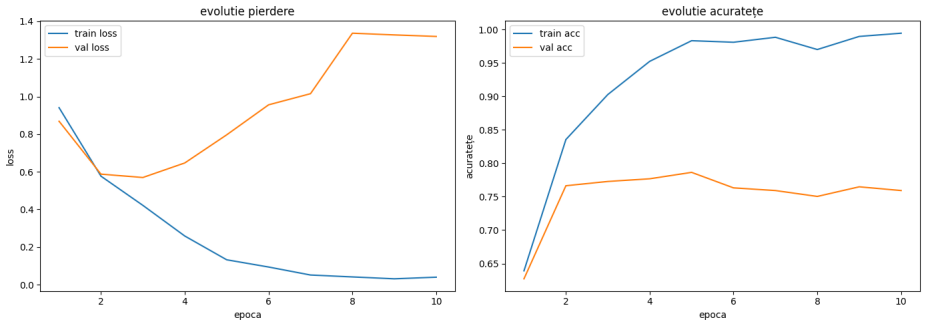

Figure 4: Evoluția metricilor pe epoci

Văzând îmbunătățirea pe datele de validare de până la 78.7%, nu am testat acest model și pe datele de test, pentru că am vrut să folosesc și sfaturile primite la curs, care includ:

- Preprocesarea imaginilor și scăderea mediei,

- Inițializarea ponderilor folosind Xavier,
- Batch Normalization,
- Optimizarea hiperparametrilor folosind încercări aleatoare.

6.2 Generare aleatoare de modele

Fișier `EnsembleRandomly.py`

6.2.1 Arhitectura modelelor

Următoarea încercare de model a fost un ansamblu de CNN-uri generate aleator, unde:

- numărul de filtre în fiecare strat convoluțional a fost ales aleator dintr-un interval definit (monotonic în capacitate);
- după fiecare convoluție am aplicat *BatchNorm2d*, *ReLU* și *MaxPool2d(2)*;
- am folosit un strat final de *AdaptiveAvgPool2d(1)*, flatten și două straturi complet conectate;
- **Optimizarea hiperparametrilor cu Optuna:**
  - Optuna este un framework de optimizare automată a hiperparametrilor care folosește, implicit, *Tree-structured Parzen Estimator* (TPE) pentru sampling și *prunare* (early stopping) a studiilor nepromițătoare.
  - La fiecare trial, Optuna alege valori pentru *l2_reg*, *dropout* şi *learning_rate*, antrenează modelul şi raportează acurateţea de validare.
  - Dacă un trial nu mai poate depăşi performanţa curentă în timpul antrenării, Optuna poate opri anticipat execuţia lui (pruning), economisind timp de calcul.
- am implementat early-stopping cu *patience*=10 și am salvat checkpoint-ul cu cea mai bună acuratețe de validare pentru fiecare trial;
- ansamblul final folosește softmax averaging peste top $N = 5$ modele.

6.2.2 Optimizatorul Adam

Optimizatorul Adam combină beneficiile *Momentum* şi *RMSProp* prin următoarele actualizări pe paşi t:

$$\begin{aligned} g_t &= \nabla_\theta \mathcal{L}(\theta_{t-1}), \\ m_t &= \beta_1\, m_{t-1} + (1-\beta_1)\, g_t, \\ v_t &= \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \\ \theta_t &= \theta_{t-1} - \alpha\, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \end{aligned}$$

unde α este *learning rate*, β_1 şi β_2 coeficienţi de decădere exponentială (implicit 0.9 şi 0.999), iar ϵ este un mic termen de regularizare.

6.2.3 Programarea ratei de învăţare prin Cosine Annealing

Cosine Annealing ajustează rata de învăţare η_t după un program:

$$\eta_t = \eta_{\min} \;+\; \tfrac{1}{2}\big(\eta_{\max} - \eta_{\min}\big)\Big(1 + \cos\!\big(\pi\, t / T_{\max}\big)\Big),$$

unde $T_{\max}$ este numărul total de epoci. În implementarea PyTorch implicită $\eta_{\min} = 0$ şi $\eta_{\max} = \alpha$.

6.2.4 Funcţia de pierdere Cross-Entropy

Pentru clasificare multi-clasă, dacă modelul prezice probabilităţi $\mathbf{p} = (p_1, \ldots, p_C)$ şi ţinta reală este vectorul one-hot $\mathbf{y}$, pierderea pe un exemplu este:

$$\mathcal{L}(\mathbf{p}, \mathbf{y}) = -\sum_{c=1}^{C} y_c\, \log p_c.$$

În practică, PyTorch combină activarea softmax $p_c = \frac{e^{z_c}}{\sum_k e^{z_k}}$ şi calculul pierderii, pentru eficienţă numerică.

6.2.5 Softmax Averaging

Fie M modele din ansamblu, fiecare prezicând $z^{(m)}(x) = \big(z_1^{(m)}(x), \ldots, z_C^{(m)}(x)\big)$ pentru C clase. Pentru fiecare model m calculăm probabilităţile prin softmax:

$$p_c^{(m)}(x) = \frac{\exp\big(z_c^{(m)}(x)\big)}{\sum_{k=1}^{C} \exp\big(z_k^{(m)}(x)\big)}.$$

Apoi, probabilitatea mediată pe ansamblu este:

$$\bar{p}_c(x) = \frac{1}{M} \sum_{m=1}^{M} p_c^{(m)}(x).$$

Predicția finală se face alegând clasa cu probabilitatea mediată maximă:

$$\hat{y}(x) = \arg\max_c \bar{p}_c(x).$$

6.2.6 Preprocesarea imaginii

În această secțiune prezentăm descrierea matematică a transformărilor aplicate pe imagine înainte de antrenare.

- *Redimensionare (Resize)*

  Fie imaginea originală $I\colon \{0,\dots,H-1\}\times\{0,\dots,W-1\} \to \mathbb{R}^3$. După redimensionare la (H', W') obținem:

  $$I'(x', y') \;=\; \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} I(i,j)\; k\Big(x'\,\frac{H}{H'} - i\Big)\; k\Big(y'\,\frac{W}{W'} - j\Big),$$

  unde k este kernel-ul de interpolare biliniară sau bicubică.

- *Transformare afină aleatorie (RandomAffine)*

  Se aplică o transformare afină:

  $$\begin{pmatrix} x'' \\ y'' \\ 1 \end{pmatrix} = T_{\text{transl}}\, T_{\text{shear}}\, T_{\text{scale}}\, T_{\text{rot}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

  cu

  $$T_{\text{rot}}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad s_x, s_y \sim \mathcal{U}(0.9, 1.1), \quad t_x, t_y \sim \mathcal{U}(-0.1, 0.1), \quad \text{shear} \sim \mathcal{U}(-$$

- *Flip orizontal aleator (RandomHorizontalFlip)*

  Cu probabilitate $p = 0.5$ se inversează coordonata orizontală:

  $$I'(x, y) = I(W - 1 - x,\, y).$$

- *Perturbări de culoare (ColorJitter)*

- **Luminozitate:** $I'_c = I_c \cdot b, \quad b \sim \mathcal{U}(0.9, 1.1)$.
- **Contrast:** $I''_c = (I'_c - 0.5)\, c + 0.5, \quad c \sim \mathcal{U}(0.9, 1.1)$.

- *Perspective aleatorie (RandomPerspective)*

  Se aplică o homografie $H \in \mathbb{R}^{3\times3}$ perturbată:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

  unde H este generată aleator cu `distortion_scale` $= 0.3$ și $p = 0.5$.

- *Conversie în tensor și normalizare*

  Imaginea devine tensor $X \in \mathbb{R}^{3\times H'\times W'}$ cu valori în $[0, 1]$. Apoi:

$$\hat{X}_{c,i,j} = \frac{X_{c,i,j} - \mu_c}{\sigma_c},$$

  unde μ_c este media canalului c calculată anterior și $\sigma_c = 1$.

- *Ștergere aleatorie (RandomErasing)*

  Cu probabilitate $p = 0.25$ se alege un dreptunghi aleator de proporție $r \sim \mathcal{U}(0.02, 0.33)$ și aspect $a \sim \mathcal{U}(0.3, 3.3)$, după care pixelii din acea regiune sunt setați la zero.

Figure 5: Schema completă a transformărilor de preprocesare aplicate unei imagini

6.2.7 Concluzie ansamblu

După o antrenare de maxim 100 de epoci și 20 patience la fiecare model (în total 17h de antrenament), acuratețea a fost de 93.3%, iar cel mai bun model a avut arhitectura de mai jos.

Layer (tip)	**Output shape**
Conv2d(3, 64, 3×3, pad=1)	$64 \times 100 \times 100$
BatchNorm2d(64)	$64 \times 100 \times 100$
ReLU	$64 \times 100 \times 100$
MaxPool2d(2×2)	$64 \times 50 \times 50$
Conv2d(64, 64, 3×3, pad=1)	$64 \times 50 \times 50$
BatchNorm2d(64)	$64 \times 50 \times 50$
ReLU	$64 \times 50 \times 50$
MaxPool2d(2×2)	$64 \times 25 \times 25$
Conv2d(64, 128, 3×3, pad=1)	$128 \times 25 \times 25$
BatchNorm2d(128)	$128 \times 25 \times 25$
ReLU	$128 \times 25 \times 25$
MaxPool2d(2×2)	$128 \times 12 \times 12$
Conv2d(128, 512, 3×3, pad=1)	$512 \times 12 \times 12$
BatchNorm2d(512)	$512 \times 12 \times 12$
ReLU	$512 \times 12 \times 12$
MaxPool2d(2×2)	$512 \times 6 \times 6$
Conv2d(512, 512, 3×3, pad=1)	$512 \times 6 \times 6$
BatchNorm2d(512)	$512 \times 6 \times 6$
ReLU	$512 \times 6 \times 6$
MaxPool2d(2×2)	$512 \times 3 \times 3$
AdaptiveAvgPool2d(1)	$512 \times 1 \times 1$
Flatten	512
Linear(512, 1024) + ReLU	1024
Dropout(0.5)	1024
Linear(1024, 5)	5

Table 1: Arhitectura CNN cea mai bună

6.3 Îmbunătățirea celui mai bun model

Fișier `BestModel.py`

Următoarea încercare importantă a fost tunarea celui mai bun model dat de alegerea random de la secțiunea anterioară. Am făcut următoarele upgrade-uri:

- Adăugarea unei ștergeri pe imagine;

  În plus față de modificările prezentate anterior, adăugăm, cu o probabilitate de 25% o ștergere astfel:

Figure 6: Schema completă a transformărilor de preprocesare aplicate unei imagini pentru actualul model

- Numărul de epoci mărit până la 300;

  Pentru că patience-ul și limitarea numărului de epoci la 100 nu aduce convergența dorită, putem ajunge la acurateţe mai mare ducând numărul de epoci la 300 și lăsând *Cosine Annealing* să își facă treaba.

- Adăugarea de *label smoothing*;

  În loc să folosim etichete one-hot pure $y_c \in \{0, 1\}$, aplicăm smoothing:

  $$y_c^{\text{ls}} = (1 - \epsilon)\, y_c \; + \; \frac{\epsilon}{C},$$

  unde C este numărul de clase și $\epsilon = 0.1$. Astfel, pierderea devine:

  $$\mathcal{L}_{\text{ls}} = -\sum_{c=1}^{C} y_c^{\text{ls}} \, \log p_c = -(1 - \epsilon) \log p_y \; - \; \sum_{c \neq y} \frac{\epsilon}{C} \log p_c.$$

  În PyTorch: `nn.CrossEntropyLoss(label_smoothing=0.1)` implementează direct această formulă pentru stabilitate numerică.

- Folosirea optimizerului *AdamW*.

  AdamW este o variantă a lui Adam cu weight decay decuplat. La fiecare pas t avem, pentru parametrii θ_t:

  $$g_t = \nabla_\theta \mathcal{L}(\theta_{t-1}), \quad m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2,$$

  $$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

  iar actualizarea este

  $$\theta_t = \theta_{t-1} - \alpha \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha \, \lambda \, \theta_{t-1},$$

  unde:

  - α este learning rate,
  - β_1, β_2 coeficienţi de moment (0.9, 0.999),

– ϵ termen de stabilitate,

– λ este weight_decay.

Implementarea PyTorch:

```
optimizer = optim.AdamW(
    model.parameters(),
    lr=1e-3,
    weight_decay=1e-4,
    eps=1e-8
)
```

6.3.1 Rezultatele modelului

Acest model a atins o acuratețe de 94.32% pe setul de validare, 94.066% pe primul 23% din setul de test și 93.2% pe restul de 77%. Acuratețea loss-ul și matricea de confuzie sunt reprezentate în imaginile de mai jos.

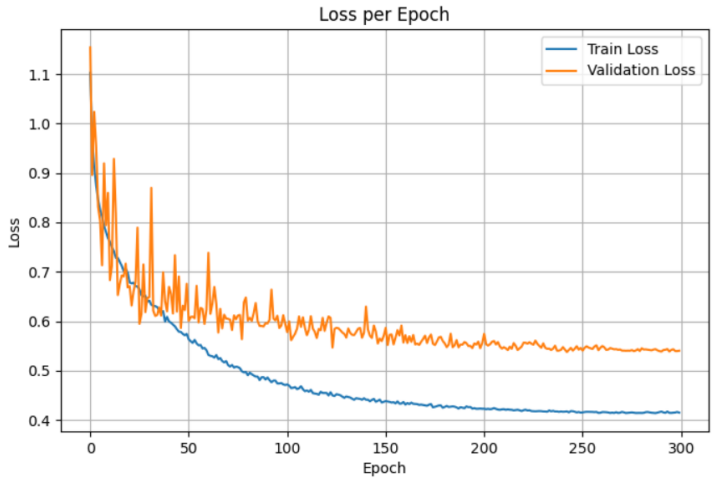

Figure 7: Loss

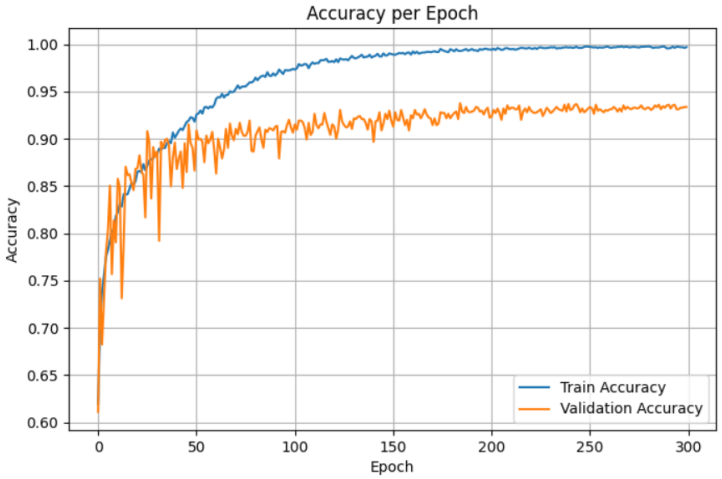

Figure 8: Acuratețea pe setul de validare

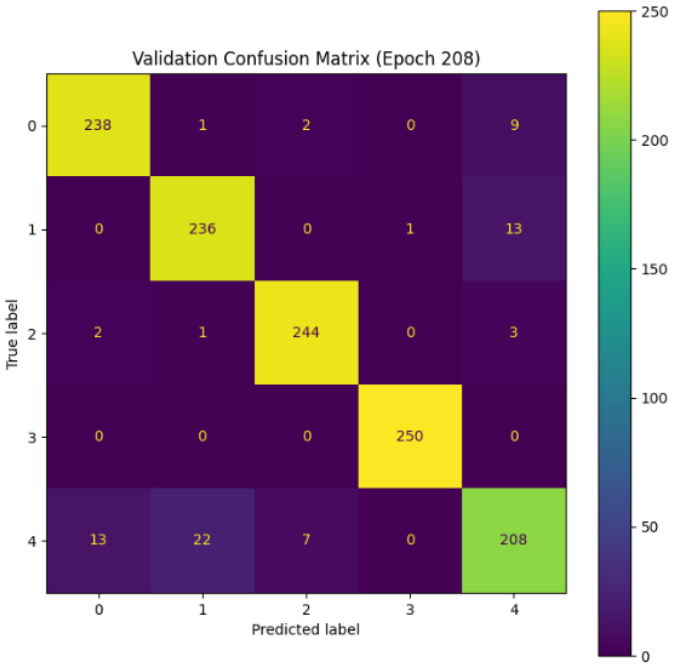

Figure 9: Matricea de confuzie pe setul de validare

Orice altă transformare pe imagini pe care am încercat-o folosind această arhitectură de model, a redus acuratețea pe setul de validare în jurul valorii de 93%, fapt ce nu reprezenta o îmbunătățire.

Am încercat și antrenarea modelului pe setul de validare, doar că nu am avut garanția acurateții, fapt ce nu a adus nicio înbunătățire, ci doar overfitting.

6.4 Ansamblu provenit dintr-un singur model

Fișier `EnsembleFromOne.py`

Ultima încercare a fost crearea unui ansamblu din top 5 cele mai bune modele din punct de vedere al acurateții pe datele de validare. Pentru generarea submisiei pe datele de test, cele 5 modele au votat label-ul pentru fiecare imagine. Label-ul pus în submisie a fost cel care a primit cele mai multe voturi, iar în caz de egalitate s-a ales label-ul dat de modelul cu cea mai mare acuratețe pe validare. Pe lângă modificarea codului de alegere a etichetei din primul ansamblu, am mai modificat modelul astfel:

- Numărul de epoci a fost setat la $E = 400$.
- Rata de învățare inițială a fost
$$\eta_0 = 3 \times 10^{-4}\,.$$
- Regularizarea prin weight decay (L2) folosește un factor
$$\lambda = 5 \times 10^{-5}\,,$$
astfel încât, la fiecare pas de optimizare, greutăţile se actualizează conform
$$w \leftarrow w - \eta\,\big(\nabla_w \mathcal{L}(w) + \lambda\, w\big).$$
- Pentru scheduling-ul ratei de învățare am folosit `CosineAnnealingWarmRestarts` cu parametrii:
$$T_0 = 10, \quad T_{\text{mult}} = 2, \quad \eta_{\min} = 10^{-8},$$
care aplică, la pasul t, formula
$$\eta_t = \eta_{\min} + \tfrac{1}{2}\,(\eta_0 - \eta_{\min})\Big[1 + \cos\!\big(\tfrac{T_{\text{cur}}}{T_i}\,\pi\big)\Big],$$
unde:

– T_i este lungimea curentă a intervalului dintre restarturi ($T_i = T_0$ pentru primul ciclu, apoi $T_i \leftarrow T_i \times T_{\text{mult}}$ după fiecare restart).

– T_{cur} este numărul de epoci scurse de la ultimul restart.

– **Restart**: după fiecare T_i epoci, scheduler-ul „repornește" ciclul de cosine annealing, adică

$$T_{\text{cur}} \leftarrow 0, \quad \eta_{T_i} = \eta_0, \quad T_i \leftarrow T_i \times T_{\text{mult}}.$$

Astfel rata de învățare revine la valoarea inițială η_0 şi se începe un nou ciclu de decădere cosinusoidală, dar pe o perioadă mai lungă.

6.4.1 Rezultatele ansamblului

Pe prima parte de 23% din setul de test, ansamblul a avut o acuratețe de 94.133%, iar pe restul de 77% a atins 93.58% acuratețe. Pe setul de validare, acuratețea a fost de 94.24%. Evoluția acurateții pe setul de antrenament și pe cel de validare se poate vedea în graficul de mai jos.

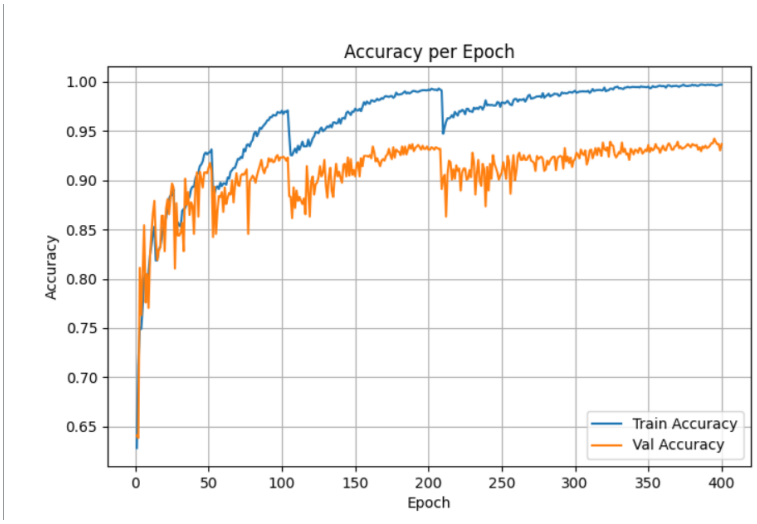

Figure 10: Acuratețe pe setul de antrenament și validare

7 Concluzie

Concluzionând, modelul cu cea mai bună acuratețe la testare a fost un ansamblu de 5 modele provenite din aceeași arhitectură, rezultatul final fiind 93.58%.